Content:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Given an array of integers, the worst-case time for each algorithm:

Insertion sort: $\mathcal{O}(n^2)$
Merge sort: $\mathcal{O}(n \log n)$
Heap sort: $\mathcal{O}(n \log n)$
Quick sort: $\mathcal{O}(n^2)$
Counting sort: $\mathcal{O}(n+k)$

1. Note 1: k can be larger than n, and it will change the complexity
2. Note 2: Won't work with real numbers or any sets S that are not countable (*i.e.* no surjective $f : \mathbb{N} \to S$) Only work on finite sets.

Radix sort: $\mathcal{O}(d(n+k))$ (Uses counting sort, so restrictions of counting sort applies)
Bucket sort: $\mathcal{O}(n^2)$ (Worst case when all numbers in the same bucket)

Visualization: `https://www.toptal.com/developers/sorting-algorithms`

Asymoptotics

def

- $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq f(n) \leq cg(n)$, $\forall n \geq n_0$
- $f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq cg(n) \leq f(n)$, $\forall n \geq n_0$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n) \Leftrightarrow \exists c_1, c_2, n_0 > 0$ s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_0$

e.g. (2022 final): What does it mean by $n! = n^n e^{-n}\sqrt{2\pi n}\left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right)$ (Stirling formula[1])?
Note here $f(n) = \frac{n!}{n^n e^{-n}\sqrt{2\pi n}} - 1$, $g(n) = \frac{1}{n}$,
So $\frac{n!}{n^n e^{-n}\sqrt{2\pi n}} - 1 \leq \frac{c}{n}$ for some $c, n_0 > 0$ and all $n \geq n_0$

properties

Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
Proof: $\exists c_1, c_2, n_1 > 0$ s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_1$.
$\exists c_3, c_4, n_2 > 0$ s.t. $0 \leq c_3 h(n) \leq g(n) \leq c_4 h(n)$, $\forall n \geq n_2$.
$\therefore 0 \leq c_1 c_3 h(n) \leq c_1 g(n)$ and $c_2 g(n) \leq c_2 c_4 h(n)$, $\forall n \geq n_2$.
$\therefore 0 \leq c_1 c_3 h(n) \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \leq c_2 c_4 h(n)$, $\forall n \geq \max(n_1, n_2)$.

Transpose: $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
Proof: $\exists c, n_0 > 0$ s.t. $0 \leq f(n) \leq cg(n)$, $\forall n \geq n_0 \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq \frac{1}{c} f(n) \leq g(n)$, $\forall n \geq n_0$

Symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
Proof: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ and $g(n) = \mathcal{O}(f(n))$

Limit Method

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$[2] (The 2 and 3 here are refering to the footnote numbers.)
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$[3]
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c \in (0, \infty) \Rightarrow f(n) = \Theta(g(n))$

L'Hopital's rule: $\lim_{x\to a} \frac{f(x)}{g(x)} = \frac{0}{0}$ or $\frac{\infty}{\infty} \Rightarrow \lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$

[2] $f(n) = o(g(n))$ if and only if $\forall c > 0$, $\exists n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
[3] $f(n) = \omega(g(n))$ if and only if $\forall c > 0$, $\exists n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

UNIVERSITY OF TORONTO

▼ Limit Method (More precisely)

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in [0,\infty) \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0,\infty) \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0,\infty] \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$

▼ log*n remark

Remark

Notation (in this course): $(\log n)^2 = (\log n)(\log n)$ and $\log^{(2)} n = \log(\log n)$
$\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$

▼ Bounded Functions

Polylogarithmically bounded: $\exists k > 0,\ f(n) = \mathcal{O}((\log n)^k)$
Polynomially bounded: $\exists k > 0,\ f(n) = \mathcal{O}(n^k)$
Exponentially bounded: $\exists k > 0,\ f(n) = \mathcal{O}(k^n)$

Polynomially-Bounded Functions

Theorem

$f(n) = \mathcal{O}(n^k) \Leftrightarrow \log(f(n)) = \mathcal{O}(\log n)$

Theorem

All Logarithmically bounded functions are polynomically bounded. i.e. $f(n) = \mathcal{O}((\log n)^a) \Rightarrow f(n) = \mathcal{O}(n^b),\ \forall a, b \geq 0$

Theorem

All polynomially bounded functions are exponentially bounded. i.e. $f(n) = \mathcal{O}(n^a) \Rightarrow f(n) = \mathcal{O}(b^n),\ \forall a > 0, b > 1$

▼ Logrithm method

Limit of logs: $\lim_{x\to a}(\log_b f(x)) = \log_b\left(\lim_{x\to a} f(x)\right)$ ($\log_b(\cdot)$ is continuous)

Suppose we want to compute $\lim_{n\to\infty}\frac{f(n)}{g(n)} = L$.

$$\log\left(\lim_{n\to\infty}\frac{f(n)}{g(n)}\right) = \log L$$

$$\lim_{n\to\infty}\left(\log\frac{f(n)}{g(n)}\right) = \log L$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = L = 2^{\lim\limits_{n\to\infty}\left(\log\frac{f(n)}{g(n)}\right)}$$

▼ Comparing Functions

Short hand notation: $f(n) << g(n) \Leftrightarrow f(n) = \mathcal{O}(g(n))$
Assume f and h are eventually positive, i.e. $\lim_{n\to\infty} f(n) > 0$ and $\lim_{n\to\infty} h(n) > 0$

$1 << \log^*(n) << \log^{(i)} n << (\log n)^a << \sqrt{n} << n << n\log n << n^{1+b} << c^n << n!$, for all positive i, a, b, c

$f(n) << g(n) \Rightarrow h(n)f(n) << h(n)g(n)$
Proof: $\lim_{n\to\infty}\frac{h(n)f(n)}{h(n)g(n)} = \lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$

$f(n) << g(n) \Rightarrow f(n)^{h(n)} << g(n)^{h(n)}$
Proof: $\log\left(\lim_{n\to\infty}\frac{f(n)^{h(n)}}{g(n)^{h(n)}}\right) = \lim_{n\to\infty} h(n)\log\frac{f(n)}{g(n)} = \lim_{n\to\infty} h\log\lim_{n\to\infty}\frac{f}{g} = -\infty$, $\lim_{n\to\infty}\frac{f^h}{g^h} = 0$

$f(n) << g(n)$ and $\lim_{n\to\infty} h(n) > 1 \Rightarrow h(n)^{f(n)} << h(n)^{g(n)}$
Proof: $\log\left(\lim_{n\to\infty}\frac{h(n)^{f(n)}}{h(n)^{g(n)}}\right) = \lim_{n\to\infty}(f-g)\log h = -\infty$, $\lim_{n\to\infty}\frac{h^f}{h^g} = 0$

▼ Simple Examples (Direct Solve)

Prove that $2^{n+1} = \mathcal{O}(2^n)$.

Solution: $0 \le 2^{n+1} = 2\cdot 2^n \le c\cdot 2^n$, $\forall n \ge 0$, if $c \ge 2$.
$\therefore$ we can choose $c = 2$, $n_0 = 1$, $0 \le 2^{n+1} \le c2^n$, $\forall n \ge n_0$.
$\therefore 2^{n+1} = \mathcal{O}(2^n)$.

Prove that $2^{n+1} = \Omega(2^n)$.

Solution: $0 \le c\cdot 2^n \le 2\cdot 2^n = 2^{n+1}$, $\forall n \ge 0$, if $c \le 2$.
$\therefore$ we can choose $c = 1$, $n_0 = 1$, $0 \le c2^n \le 2^{n+1}$, $\forall n \ge n_0$.
$\therefore 2^{n+1} = \Omega(2^n)$.

Proof Methods (Induction, Contradiction, & Combinatorial Arguments)

- **Weak Induction**
  - def

    Weak induction:
    1. Basis: show $P(n_0)$ is true
    2. Hypothesis: Assume $P(n)$ is true (**!!!Note:** You should not assume it is true for all n. This is what you need to prove. Assume $P(n)$ is true for all n will cost you 2-3 marks in exams.)
    3. Induction: Show $P(n) \Rightarrow P(n+1)$
  - simplest example

    Prove $n! \leq n^n$, $\forall n \geq 1$
    Proof: **Base Step**: $n = 1$, $1! = 1 = 1^1$
    (i) Using n only:
    Induction Hypothesis: Assume $n! \leq n^n$
    Induction Step: For $n+1$, $(n+1)! = (n+1)n! \leq (n+1)n^n$ by IH
    $\leq (n+1)(n+1)^n = (n+1)^{n+1}$
  - simplest example: use k (same thing as previous)

    (ii)Introducing a new variable k:
    Induction Hypothesis: Assume when $n = k \geq 1$, $k! \leq k^k$
    Induction Step: When $n = k+1$, $(k+1)! = (k+1)k! \leq (k+1)k^k$ by IH
    $\leq (k+1)(k+1)^k = (k+1)^{k+1}$

Weak induction examples

Prove $11^n - 6$ is divisible by 5, $\forall n \geq 1$
Let $P(n) = 5|(11^n - 6)$ (5 divides $11^n - 6$)
Base Step: $n = 1$, $11^1 - 6 = 5$ $5|5$
Induction Hypothesis: Assume $P(n)$ is true
Induction Step: Check $P(n+1)$, $11^{n+1} - 6 = 11 \cdot 11^n - 6 = 11 \cdot (5m + 6 - 6)$
$= 55m + 66 - 6 = 55m + 60 = 5(11m + 12)$ for some m
So $5|11^{n+1} - 6$, $P(n+1)$ is true.

Strong Induction

1. Basis: show $P(n_0), P(n_1), \ldots$ are true
2. Hypothesis: Assume $P(k)$ is true, $\forall k \leq n$
3. Induction: Show $P(n_0) \wedge \cdots \wedge P(k) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$

e.g. The Fundamental Theorem of Arithmetic: all integers $n \geq 2$ can be expressed as the product of one or more prime numbers
Proof: **Base Step**: $n = 2$, 2 is a prime
Induction Hypothesis: assume all $k \in [2, n]$ can be written as the product of one or more primes
Induction Step:
$n+1$ is prime. Then it can be expressed as the product of itself.
$n+1$ is not prime. Then $n+1 = k_1 k_2$ for some integers $k_1, k_2 < n+1$. By IH, k_1, k_2 can be written as product of primes. Thus $n+1$ can be written as product of primes

Prove that using \$2 and \$5, we can make any amount $\geq$ \$4
Proof: **Base Step**: $n = 4$ can be made using \$2+\$2, $n = 5$ can be made using \$5
Induction Hypothesis: assume $\forall k \in [4, n]$, \$$k$ can be made using \$2 and \$5
Induction Step: $(n+1) = (n-1) + 2$ and $n-1$ can be made using \$2 and \$5 by IH

Prove by Contradiction

1. Assume toward a contradiction $\neg P$/ $\bar{P}$(not P)
2. Make some argument
3. arrive at a contradiction
4. $\therefore P$ must be true

e.g. Prove that there are infinitely many prime number
Proof: Assume that there are a finite number of primes
Let S be the complete set of primes
let $P = \prod_{x \in S} x + 1$
$P \notin S$ because $P > x$, $\forall x \in S$
P is a prime because P is not divisible by any prime, $1 \equiv P \mod x$, $\forall x \in S$
P is a prime but not in S, contradiction.

Recurrence:

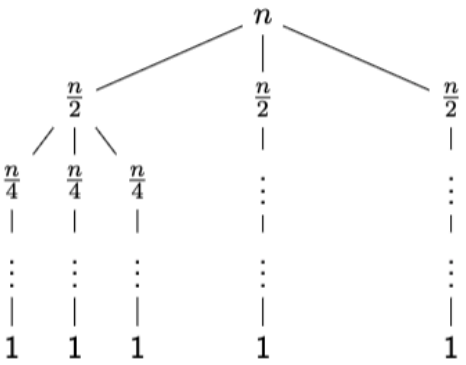
Recurrence:

Find asymptotic expressions for the following $T(n)$:
e.g. $T(n) = T(n-1) + n$, $T(1) = 1$
Proof: $T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \cdots = T(1) + 2 + 3 + 4 + \cdots + (n-1) + n$
$T(n) = \sum_{i=1}^{n} = \frac{n(n+1)}{2}$, $T(n) = \Theta(n^2)$

e.g. $T(n) = T(n/2) + n$, $T(1) = 1$
Proof: $T(n) = T(n/2) + n = T(n/4) + n/2 + n = T(n/8) + n/4 + n/2 + n = \cdots =$
$T(1) + 2 + 4 + 8 + \cdots + (n/4) + (n/2) + n$
$T(n) = \sum_{i=0}^{\log n} \frac{n}{2^i} = n\frac{1 - (1/2)^{\log n + 1}}{1 - 1/2} = n(2 - 1/n) = 2n - 1$, $T(n) = \Theta(n)$

Basic examples

Find asymptotic expressions for the following $T(n)$:
e.g. $T(n) = 3T(n/2) + n$, $T(1) = 1$
Proof: $T(n) = 3T(n/2) + n = 3(3T(n/4) + n/2) + n = 3^2T(n/4) + 3(n/2) + n = \cdots = 3^kT(1) + 3^{k-1}(n/2^{k-1}) + \cdots + 3^2(n/2^2) + 3(n/2) + n = 3^kT(1) + \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i n$
Assume $n = 2^k$, $k = \log n$ $T(n) = 3^{\log n} + \sum_{i=0}^{\log n - 1} \left(\frac{3}{2}\right)^i n = n^{\log 3} + n\frac{(3/2)^{\log n} - 1}{(3/2) - 1} =$
$n^{\log 3} + 2n(n^{\log 3}/n - 1) = 3n^{\log 3} - 2n$,
$T(n) = \Theta(n^{\log 3})$



Master Theorem:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1, b \geq 1$, $f(n)$ is asymptotically positive

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$, $\epsilon > 0$. Then $T(n) = \Theta(n^{\log_b a})$ (cost of solving the sub-problems at each level increases by a certain factor, the last level dominates)
- Case 2: $f(n) = \Theta(n^{\log_b a})$. Then $T(n) = \Theta(n^{\log_b a} \log n)$ (cost to solve subproblem at each level is nearly equal)
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and $n > n_0$ (regularity condition, always holds for polynomials). Then $T(n) = \Theta(f(n))$ (cost of solving the subproblems at each level decreases by a certain factor)

Method:

1. identify a and b and compute $\log_b a$
2. compare $n^{\log_b a}$ to $f(n)$ and decide which case applies
3. don't forget to check the regularity condition for case 3

UNIVERSITY OF TORONTO

Examples

$T(n) = 7T\left(\frac{n}{2}\right) + n^2$ (Works for $n^2 \log n$)
Solution: $a = 7, b = 2, \log_2 7 \approx 2.8, f(n) = n^2 = \mathcal{O}(n^{\log_2 7 - \epsilon})$ for $\epsilon \in (0, \log_2 7 - 2)$
Case 1, $T(n) = \Theta(n^{\log_2 7})$

$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
Solution: $a = 4, b = 2, \log_2 4 = 2, f(n) = n^{2.5} = \Omega(n^{2+\epsilon})$ for $\epsilon \in (0, 0.5]$
Case 3, Check regularity: $af\left(\frac{n}{b}\right) = 4\left(\frac{n}{2}\right)^{2.5} \leq cn^{2.5}$. Choose $c \in \left[\frac{1}{\sqrt{2}}, 1\right)$
$T(n) = \Theta(f(n)) = \Theta(n^{2.5})$

$T(n) = T(\sqrt{n}) + 1$
Solution: let $n = 2^m$, $S(m) = T(2^m) = T(n)$
Then $T(2^m) = T(2^{\frac{m}{2}}) + 1 \Leftrightarrow S(m) = S\left(\frac{m}{2}\right) + 1$
$a = 1, b = 2, \log_2 1 = 0, f(m) = 1 = m^0 = \Theta(m^0)$
Case 2, $T(2^m) = S(m) = \Theta(\log m)$, $T(n) = \Theta(\log \log n)$

Heaps & Heapsort:

Heap Definition:

Definition: A heap is an array $A = [a_1, a_2, ..., a_n]$ of elements such that:

- Heap shape property: the heap is an almost complete binary tree (all except the last row is full)
- Heap order property:
  - Max-heap: $\forall i,\ A[\text{parent}(i)] \geq A[i]$
  - Min-heap: $\forall i,\ A[\text{parent}(i)] \leq A[i]$

Indexing parents and children (Assume index of array starts at 1)

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

Heap Properties:

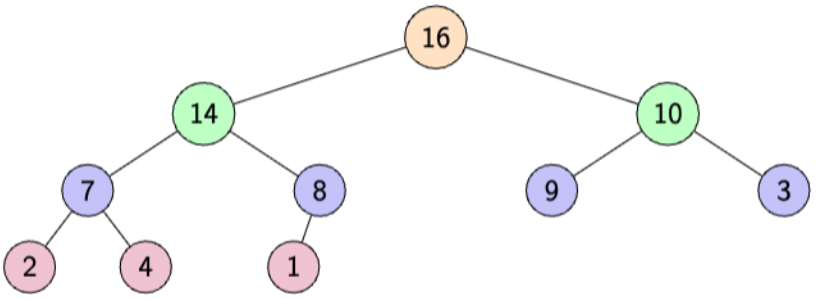
Key Take-Aways

- $2^h \leq n \leq 2^{h+1} - 1 \Leftrightarrow h = \lfloor \log n \rfloor$
- For a max-heap, the maximum value will always occur at the root
- For a min-heap, the minimum value will always occur at the root
- Heapsort is $\Theta(n \log n)$ in time and $\Theta(1)$ in space (space complexity only considers extra memory needed excluding input)
- Heapsort is an in-place algorithm
- Heapsort is not stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Example

$A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$



Basic proofs

Let h be the height and n the number of nodes in a heap, $2^h \leq n \leq 2^{h+1} - 1$
Proof: the number of elements in a full binary tree of height h is $2^{h+1} - 1$. Min occurs if only 1 element in the last level.

Let h be the height and n the number of nodes in a heap, $h = \lfloor \log n \rfloor$
Proof: $2^h \leq n \leq 2^{h+1} - 1 \Rightarrow h \leq \log n < h + 1 \Rightarrow h = \lfloor \log n \rfloor$ since h is integer

In a max-heap, the root contains the largest value
Proof by induction:
Base Step: $h = 0$, root is the only element
Induction Hypothesis: Assume the propsition is true for heaps of size $0 < k < n$
Induction Step: Consider the left and right subtree of the root, by IH, their roots are the maximum of the subheap, then by max-heap property, the root is larger than all its children.

Heap Sort:

操作	作用	单次耗时	总耗时
Max-Heapify(A, i)	让以 `i` 为根的子树恢复最大堆性质	`O(h)`，h = 子树高度 ≤ `O(log n)`	—
Build-Max-Heap(A)	从无序数组建立最大堆	`O(n)`	—
Extract-Max(A)	取出堆顶最大元素并重新调整堆	`O(log n)`	`O(n log n)`（在排序中执行 n 次）
HeapSort(A)	先建堆，再不断取出最大元素	`O(n log n)`	`O(n log n)`
Insert(A, key)	插入一个新元素到堆中	`O(log n)`	—
Increase-Key(A, i, key)	增大节点值并向上调整	`O(log n)`	—
Find-Max(A)	取堆顶元素，不改动	`O(1)`	—

Max-Heapify:

- **Max-Heapify** 只“直接操作”以 **i** 为根的当前层和它的下一层（父节点与两个孩子），
- 但通过递归调用，它间接修复整个以 **i** 为根的整棵子树。

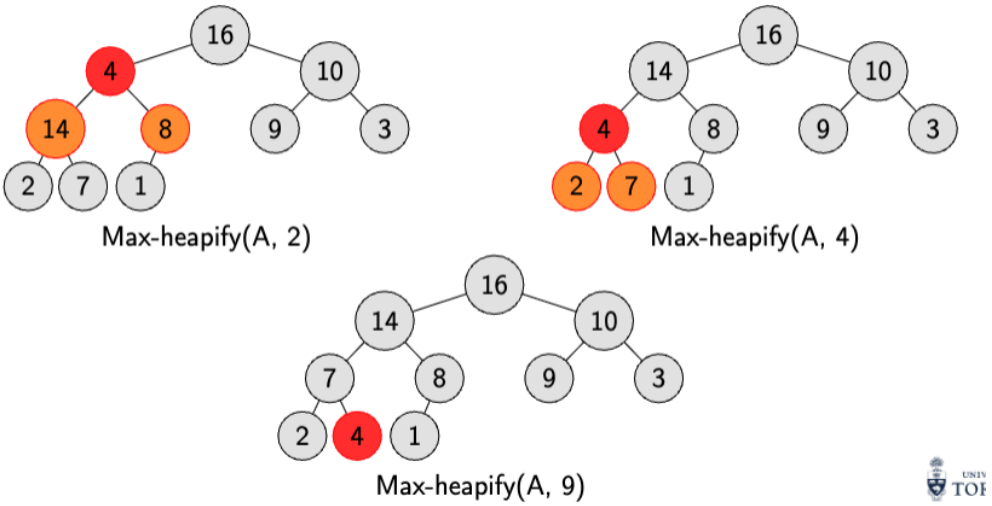
Max-Heapify

Enforces the heap order property if it is violated

1: **function** Max-Heapify(A, i)
2: Compare $A[i]$ with $A[2i]$ and $A[2i+1]$
3: **if** $A[i]$ is smaller **then**
4: swap $A[i] \leftrightarrow \max(A[2i], A[2i+1])$
5: Recurse downwards, until property is not violated or hit a leaf node

Runtime: $\mathcal{O}(h) = \mathcal{O}(\log n)$

Example



Built Max Heap:

把一个“乱序数组”一步步改造成合法的最大堆（Max-Heap）

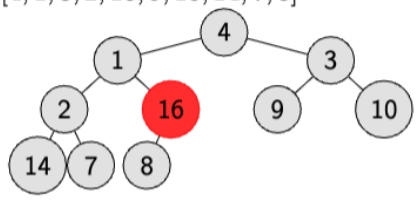
1: **function** Build-Max-Heap(A, n)
2: **for each** $i = \lfloor \frac{n}{2} \rfloor : 1$ **do** ▷ the rest are leaf nodes
3: Max-Heapify(A, i)

Runtime:

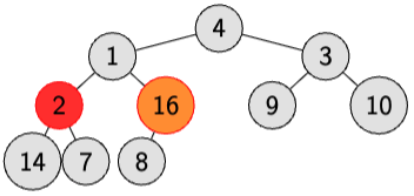
- Simple: $\mathcal{O}(n \log n)$ (for loop $\times$ cost for Heapify)
- Actual: Time to run Max-Heapify is linear in the height of the node it is run on and most node have smaller height
  - At height h, there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes.
  - height of heap is $\lfloor \log n \rfloor$
  - Runtime: $\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \mathcal{O}(h) \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} = \cdots = \mathcal{O}(n)$

Example

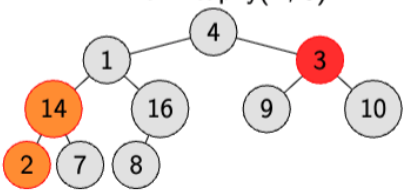
$A = [4, 1, 3, 2, 16, 9, 10, 14, 7, 8]$



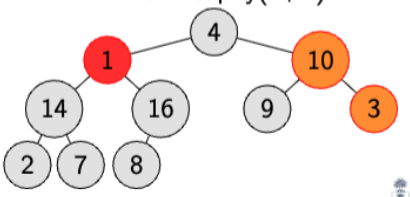
Max-heapify(A, 5)



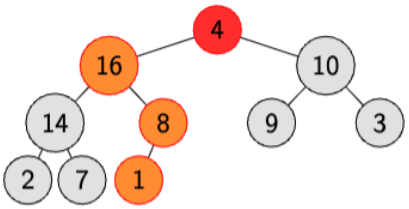
Max-heapify(A, 4)



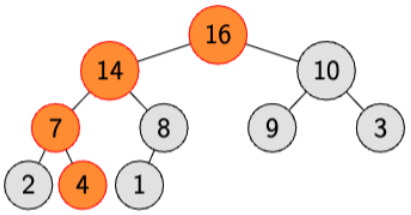
Max-heapify(A, 3)



Max-heapify(A, 2)



Max-heapify(A, 1)



Final result: $A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$

Heapsort:

Build-Max-Heap 只是 Heapsort 的前半步，它的目标是“构造堆”；
而 Heapsort 是一个完整的排序算法，在建堆之后还会“反复取最大值、放到末尾”——那才是排序。

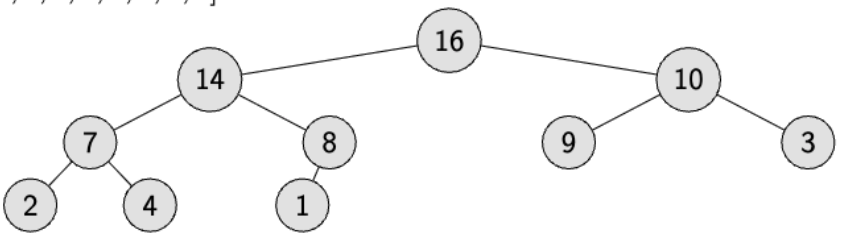
```
1: function Extract-Max(A)
2:     Swap A[1] ↔ A[A.size()]
3:     A.size = A.size-1
4:     Max-Heapify(A, 1)
```

```
1: function Heapsort(A, n)
2:     Build-Max-Heap(A, n)
3:     for each i = n : 2 do
4:         Extract-Max(A)
```

Runtime: $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

Example

$A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$



Iteration 1: $A = [14, 8, 10, 7, 1, 9, 3, 2, 4|16]$
Iteration 2: $A = [10, 8, 9, 7, 1, 4, 3, 2|14, 16]$
Iteration 3: $A = [9, 8, 4, 7, 1, 2, 3|10, 14, 16]$
Iteration 4: $A = [8, 7, 4, 3, 1, 2|9, 10, 14, 16]$
Iteration 5: $A = [7, 3, 4, 1, 2|8, 9, 10, 14, 16]$
Iteration 6: $A = [4, 3, 1, 2|7, 8, 9, 10, 14, 16]$
Iteration 7: $A = [3, 2, 1|4, 7, 8, 9, 10, 14, 16]$
Iteration 8: $A = [2, 1|3, 4, 7, 8, 9, 10, 14, 16]$
Iteration 9: $A = [1|2, 3, 4, 7, 8, 9, 10, 14, 16]$
Iteration 10: $A = [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]$

Quick Sort:

https://www.bilibili.com/video/BV1dmzHYZEBp/?spm_id_from=333.337.search-card.all.click&vd_source=3717cb523f98d837907369c5ba2ca663
https://www.bilibili.com/video/BV114411c7D6/?spm_id_from=333.337.search-card.all.click&vd_source=3717cb523f98d837907369c5ba2ca663

Quicksort

Key Take-Aways

- Worst case run time: $\Theta(n^2)$ (the list is already sorted)
  - pivot always the largest/smallest. Everytime we get one empty array and one array of size $n-1$. $T(n) = T(n-1) + \mathcal{O}(n)$
- Best case run time: $\Theta(n \log n)$
  - pivot always median, $T(n) = 2T(n/2) + \mathcal{O}(n)$
- Average/Expected run time: $\Theta(n \log n)$
  - $T(n) = T(an) + T(bn) + \mathcal{O}(n)$, where $a + b = 1$ as we have to run through the full array.
- Quicksort tends to have the smallest constant in front of its runtime
- Quicksort is not stable (although can modify to be stable)
- Quicksort is in-place (although recursion stores stuff on stack, based on exact definition of in-place)
- The idea of a randomized algorithm. Why randomization helps and how to analyze a random algorithm

UNIVERSITY OF TORONTO

Quicksort

```
1: function Partition(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for each j = p : r − 1 do
5:         if A[j] < x then
6:             i = i + 1
7:             Swap A[i] ↔ A[j]
8:     Swap A[i + 1] ↔ A[r]
9:     Return i + 1
10: function Randomized-Partition(A, p, r)
11:     i = RAND(p, r)
12:     Swap A[i] ↔ A[r]
13:     Return Partition(A, p, r)
```

```
1: function Quicksort(A, l, r)        ▷ Checking A[l, ..., r]
2:     if l < r then                  ▷ Otherwise no subarray
3:         p =Partition(A, l, r)      ▷ Split the array by p
4:         Quicksort(A, l, p − 1)     ▷ Recurse on ≤ p
5:         Quicksort(A, p − 1, r)     ▷ Recurse on > p
```

If we randomly shuffle input or choose pivot, we reduce the chance of getting the worst case scenario.
The worst case scenario is still $\mathcal{O}(n^2)$, but the chance is lower.

Partition

The array is separated into 4 parts [pivot | $\leq$ pivot | $>$ pivot | unanalyzed]
At each iteration, examine an element in the unanalyzed part
Compare to pivot and put in the appropriate section
$[\boxed{26}|||33, 35, 29, 19, 12, 22]$
$[\boxed{26}||33, 35, 29|19, 12, 22]$ $(19 \leq 26)$
$[\boxed{26}|19|33, 35, 29|12, 22]$ $(12 \leq 26)$
$[\boxed{26}|19, 12|33, 35, 29|22]$ $(22 \leq 26)$
$[\boxed{26}|19, 12, 22|33, 35, 29|]$
Swap
$[22, 19, 12, \boxed{26}, 33, 35, 29]$

Sorting and Searching in Linear time

Counting Sort:

Counting Sort

Key Take-Aways

- Counting sort assumes elements are integers ranging from 0 to k
- Runtime: $\Theta(n + k)$ (if $k = \mathcal{O}(n)$, then runtime is $\Theta(n)$)
- Counting sort uses no comparisons (it uses values of elements to determine the position)
- Counting sort is not in-place
- Counting sort is stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Algorithm

Input: $A[1, ..., n]$, $A[j] \in \{1, ..., k\}$
Output: $B[1, ..., n]$ sorted
Create an auxiliary array $C[1, ...k]$, recording the number of elements in A with value $\leq x$

```
1: function Counting-Sort(A[1, .., n], B[1, ..., n], n, k)
2:     for each i = 0 : k do
3:         C[i] = 0
4:     for each j = 1 : n do
5:         C[A[j]] = C[A[j]] + 1          ▷ # elements in A with value x
6:     for each i = 1 : k do
7:         C[i] = C[i] + C[i − 1]         ▷ # elements in A with value ≤ x
8:     for each j = n : 1 do              ▷ iterate A backwards
9:         B[C[A[j]]] = A[j]              ▷ C[A[j]] implies the sorted position of A[j]
10:        C[A[j]] = C[A[j]] − 1
```

假设要排序数组：

```ini
A = [4, 2, 2, 8, 3, 3, 1]
```

Step 1️⃣ 找出最大最小值

最大值 `max = 8`
最小值 `min = 1`

范围长度：`k = 8 - 1 + 1 = 8`

Step 2️⃣ 建立计数数组 `C[0..k-1]`

初始化全为 0：

```makefile
C = [0, 0, 0, 0, 0, 0, 0, 0]
       ↓
       对应原数组中的值 1~8
```

Step 3️⃣ 统计每个元素出现次数

扫描 `A`：

元素	操作
4	C[3]++ → [0,0,0,1,0,0,0,0]
2	C[1]++ → [0,1,0,1,0,0,0,0]
2	C[1]++ → [0,2,0,1,0,0,0,0]
8	C[7]++ → [0,2,0,1,0,0,0,1]
3	C[2]++ → [0,2,1,1,0,0,0,1]
3	C[2]++ → [0,2,2,1,0,0,0,1]
1	C[0]++ → [1,2,2,1,0,0,0,1]

Step 4️⃣ 累加计数数组（Prefix Sum）

让 `C[i]` 表示"小于等于 i 的元素个数"。

```ini
C = [1, 3, 5, 6, 6, 6, 6, 7]
```

解释：

- 有 1 个 ≤1 的数
- 有 3 个 ≤2 的数
- 有 5 个 ≤3 的数
- 有 6 个 ≤4 的数
- ...
- 有 7 个 ≤8 的数（全部）

Step 5️⃣ 填充输出数组 B

从原数组的 **最后一个元素往前** 遍历（保证稳定性）：

元素	查C[]	放入B[]	更新C[]
1	C[0]=1	B[1]=1	C[0]=0
3	C[2]=5	B[5]=3	C[2]=4
3	C[2]=4	B[4]=3	C[2]=3
8	C[7]=7	B[7]=8	C[7]=6
2	C[1]=3	B[3]=2	C[1]=2
2	C[1]=2	B[2]=2	C[1]=1
4	C[3]=6	B[6]=4	C[3]=5

得到：

```ini
B = [1, 2, 2, 3, 3, 4, 8]
```

✅ 排序完成！

Radix Sort:

Radix Sort

Key Take-Aways

- Radix sort assumes all elements have $\leq d$-digits
- Runtime $\Theta(d(n+k))$ for d-digit numbers and each digit $\in [0, k]$
- Runtime $\Theta\left(\frac{b}{r}(n+2^r)\right)$ for b-bit numbers and $r = \min(b, \lceil \log n \rceil)$
- Overall, it sorts in $\Theta(n)$
- Radix sort uses no comparisons
- Radix sort is not in-place
- Radix sort is stable

Algorithm

```
1: function Radix-Sort(A, d)
2:     for each i = 1 : d do                    ▷ Sort least sig digit first
3:         Stable sort A on digit i    ▷ Relative order in previous step is preserved
```

Example

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

7	2	0
3	5	3
4	3	6
4	5	7
6	5	7
3	2	9
8	3	9

7	2	0
3	2	9
4	3	6
8	3	9
3	5	5
4	5	7
6	5	7

3	2	9
3	5	5
4	3	6
4	5	7
6	5	7
7	2	0
8	3	9

Runtime

Lemma

Given n-bit numbers, and any positive integer $r \leq b$. Radix sort sorts in $\Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$, if stable sort is $\Theta(n+k)$.

Let $b = \#$ bits per element, $r = \#$ bits per word, $d = \#$ digits per word, $k = \#$ possible values per digit.
If $d = \left\lceil \frac{b}{r} \right\rceil$ and $k = 2^r$, then $(b, r) \Leftrightarrow (d, k)$

Choose the base $d = \left\lceil \frac{b}{r} \right\rceil$ to use for radix sort
e.g. if $b = 16$, and $r = 2$, we can make $k = r^2 = 2^2 = 4$ possible values and $d = \lceil \frac{16}{2} \rceil = 8$

Choose r to minimize the run time
Intuition: We need to balance the terms $\frac{b}{r}$ and 2^r
If r large, $\frac{b}{r}$ is small and 2^r is large
If r small, $\frac{b}{r}$ is large and 2^r is small

For $\Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$ to be $\Theta(n)$, we need $\frac{b}{r} = \mathcal{O}(1)$ and $2^r = \mathcal{O}(n)$
$r = \mathcal{O}(b)$ and $r = \mathcal{O}(\log n)$, so $r = \mathcal{O}(\min(b, \log n))$

UNIVERSITY OF TORONTO

📘 二、符号说明

符号	含义
`b`	每个元素的总位数（bit 数）
`r`	每次处理的位数（即 radix 的大小）
`d = ⌈b / r⌉`	总共需要多少趟排序（多少"位组"）
`k = 2^r`	每组位的取值范围（即计数排序的桶数）

例如：

- 如果 `b = 16`，`r = 2`，
  那么 `d = 8`，每次 2 位，一共分 8 组；
  每组可能的取值是 `2^2 = 4` 种（00, 01, 10, 11）。

⚙️ 三、Radix Sort 的时间复杂度推导

Radix Sort 的每一趟（对一组 r 位的排序）要花：

> Θ(n + k) 时间
> （因为内部使用 Counting Sort）

而总共有 `d = b/r` 趟。
所以总时间为：

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

🧱 五、找到最佳 r

我们希望：
1️⃣ $\frac{b}{r} = O(1)$，也就是 r 不太小；
2️⃣ $2^r = O(n)$，也就是 r 不太大。

第二个式子 $2^r = O(n)$ → 得出

$$r = O(\log n)$$

结合起来：

$$r = O(\min(b, \log n))$$

✅ 六、最终结论

> 当 `r = θ(log n)` 时，Radix Sort 的时间复杂度为：

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

换句话说：

- 对"字长 b 固定"的机器（例如 b = 32 位整数）
  → Radix Sort = **线性时间 O(n)**。
- 对超大数（b 很大）
  → 时间 = Θ((b/log n) * n)。

这一页解释了 **Radix Sort 的运行时间在不同输入规模下（b 与 log n 的关系）** 的两种情况，说明了当数字位数多或少时，最优的 `r`（每次处理的位数）不同，
最终得出结论：

$$T(n) = \begin{cases} \Theta(n), & \text{if } b < \log n, \\ \Theta\left(\frac{bn}{\log n}\right), & \text{if } b \geq \log n. \end{cases}$$

BSTs

Definition

BST properties:

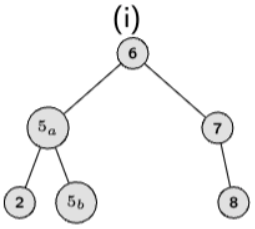
- If y is in the left subtree of x, then key$[y] \leq$ key$[x]$
- If y is in the right subtree of x, then key$[y] \geq$ key$[x]$

Key Take-Aways

- In general, if a BST has n nodes, $h = \mathcal{O}(n)$. Only if the BST is balanced, $h = \mathcal{O}(\log n)$
- Minimum/Maximum and searching for any arbitrary key $\mathcal{O}(h)$
- Successor/Predecessor $\mathcal{O}(h)$
- Insertion/Deletion $\mathcal{O}(h)$
- Build-BST: $\mathcal{O}(n^2)$ worst case (chain)

Basic operations

(i)



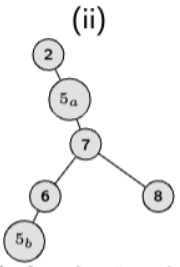

```
1: function In-Order(x)
2:     if x ≠NIL then
3:         In-Order(x.left)
4:         Process x
5:         In-Order(x.right)
```

(i)2, 5_a, 5_b, 6, 7, 8
(ii)2, 5_a, 5_b, 6, 7, 8
Used for sorting

```
1: function Pre-Order(x)
2:     if x ≠NIL then
3:         Process x
4:         Pre-Order(x.left)
5:         Pre-Order(x.right)
```

(i)6, 5_a, 2, 5_b, 7, 8
(ii)2, 5_a, 7, 6, 5_b, 8
Used for rotation

(ii)




```
1: function Post-Order(x)
2:     if x ≠NIL then
3:         Post-Order(x.left)
4:         Post-Order(x.right)
5:         Process x
```

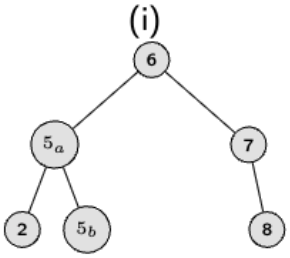
(i)2, 5_b, 5_a, 8, 7, 6
(ii)5_b, 6, 8, 7, 5_a, 2
Used for deleting

$\mathcal{O}(n)$, since we process each node exactly once

UNIVERSITY OF TORONTO

Basic operations

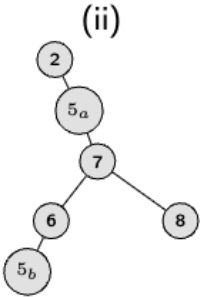
(i)




```
1: function Minimum(x)
2:     while x.left ≠ NIL do
3:         x = x.left
4:     Return x
```

(i)$6 \to 5_a \to 2$
(ii)2

(ii)




```
1: function Maximum(x)
2:     while x.right ≠ NIL do
3:         x = x.right
4:     Return x
```

(i)$6 \to 7 \to 8$
(ii)$2 \to 5 \to 7 \to 8$

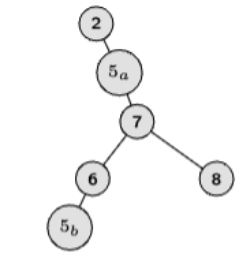
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

Predecessor(x) 是指 在二叉搜索树（**BST**）中，键值比 **x** 小的节点里最大的一个。

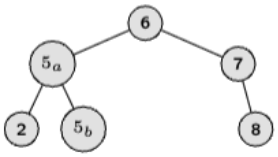
Basic operations

```
1: function Predecessor(x)
2:     if x.left ≠ NIL then
3:         Return Maximum(x.left)
       y =parent(x)
4:     while y ≠ NIL and x == y.left do
5:         x = y
6:         y =parent(y)
7:     Return y
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height



Predecessor$(5_b) = 5_a$



Predecessor$(6) = 5_b$

Predecessor(x) 是 BST 中比 x 小的最大节点。
如果有左子树 → 左子树中最大节点；
如果没有左子树 → 向上找第一个让你在“右边”的祖先。

Insert

```
1: function Insert(T, z)
2:     y=NIL, x = T.root
3:     while x ≠NIL do
4:         y = x                    ▷ Find where z should connect
5:         if z.key < x.key then
6:             x = x.left
7:         else
8:             x = x.right
9:     parent(z)=y
10:    if y==NIL then
11:        T.root = z
12:    else if z.key < y.key then
13:        y.left=z
14:    else
15:        y.right=z
```

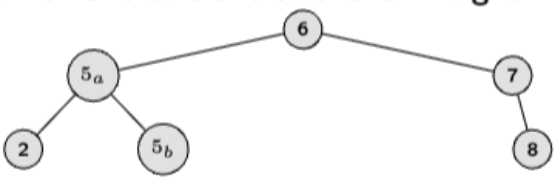
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

Build a BST from $[6, 5, 7, 5, 8, 2]$

Search

```
1: function Search(x, k)
2:     if x==NIL or x.key == k then
3:         Return x
4:     if x.key > k then
5:         Return Search(x.left, k)
6:     if x.key < k then
7:         Return Search(x.right, k)
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height



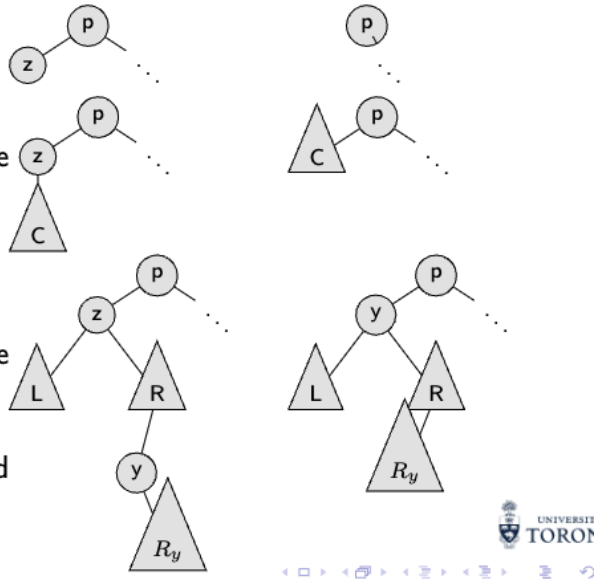
Delete

Successor(x) 是指 在二叉搜索树（BST）中，键值比 x 大的节点里最小的一个。

Case 1: z is a leaf, delete it.

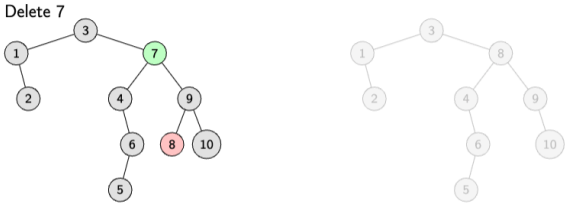
Case 2: z has one child, delete z and replace with child.

Case 3: z has 2 children.
Let y be z's successor, y never has a left subtree by defintion.
If y has no children, replace z with y
If y has a right subtree, replace z with y and replace y with right child



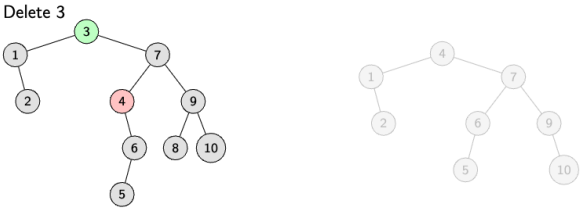
Example

Delete 7



Example

Delete 3



Example

Show if a BST node has 2 children, then the successor has no left child and the predecessor has no right child

Proof: Assume that node x has 2 children, and its successor s (minimum element of the BST rooted at x.right) has left child l, s.key $>$ l.key $>$ x.key, then l is the successor of x. Contradiction
Similarly, if its predecessor p (maximum element of the BST rooted at x.left) has right child r, we have x.key $>$ r.key $>$ p.key, r is the predecessor of x. Contradiction.

BST-sort works by constructing a BST out of the array and then calling In-Order traversal. What is the best and worst case?

Solution: In-Order always take $\Theta(n)$

Worst case: array is already sorted, BST is a linked list

Each BST at position i insert takes $\mathcal{O}(i)$, summing up gives $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2)$

Best case: BST is always perfectly balanced, height is always $\mathcal{O}(\log i)$

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

Red-Black Trees (RBTs)

左 < 根 < 右；根&叶 黑色；红色不连续；各路径黑色数量相同。

Defintion

Red Black Tree is a Binary Search Tree with the following additional properties:

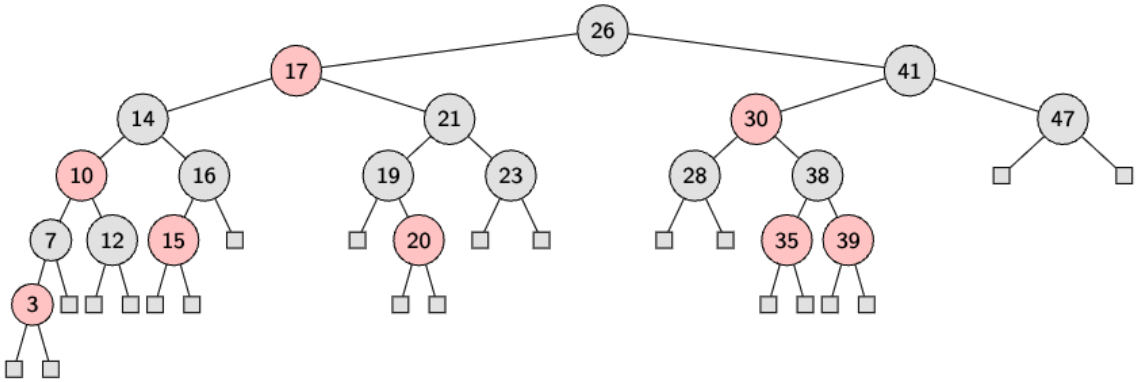
- Every node is either red or black
- The root is black
- All leaves (NIL) are black
- If a node is red, both its children are black
- For all nodes, all paths to all leaves have the same black-height (number of black nodes on path to a leaf, not including itself, but including NIL)

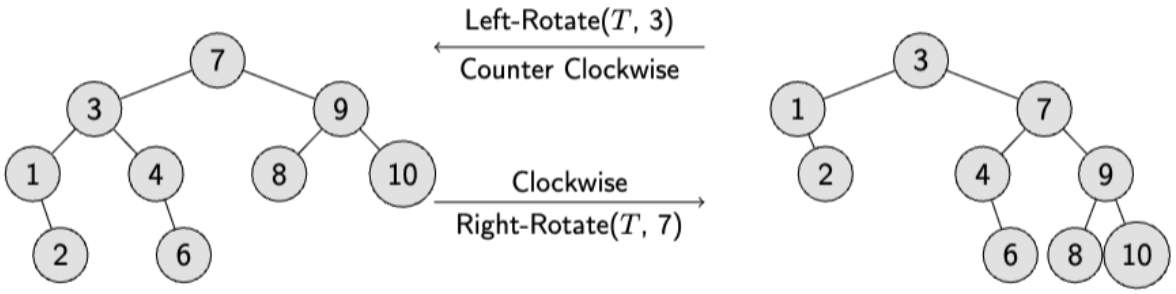
Key Take-Aways

- RB Treees are balanced $h = \mathcal{O}(\log n)$
- All read-only operations (e.g. traversals) are all the same as BST
- Rotation

UNIVERSITY OF TORONTO

Example





Hashing

(a) Demonstrate the insertion of keys 8, 15, 42, 17, 30, 27, 26, 33, 14, 24 into a doubly-hashed table where collisions are resolved with open addressing. Let the table have 11 slots, let the *primary* hash function be $h_p(k) = k \mod 11$ and let the *secondary* hash function be $h_s(k) = 3k \mod 4$.

2(a)

∵ 11 slots ∴ By the requirement, address is from 0 to 10

Put numbers by using double hashing and open addressing as defined

8: $hp(8) = 8 \mod 11 = 8$ → no collision → put 8

15: $hp(15) = 15 \mod 11 = 4$ → no collision → put 4

42: $hp(42) = 42 \mod 11 = 9$ → no collision → put 9

17: $hp(17) = 17 \mod 11 = 6$ → no collision → put 6

30: $hp(30) = 30 \mod 11 = 8$ → collision with $hp(8)$

→ $hs(30) = 3 \times 30 \mod 4 = 2$

→ 1st explore: $hp(8) + 1 \cdot hs(8) = 8 + 2 = 10$ → no collision → put 10

27: $hp(27) = 27 \mod 11 = 5$ → no collision → put 5

26: $hp(26) = 26 \mod 11 = 4$ → collision with $hp(15)$

→ $hs(26) = 3 \times 26 \mod 4 = 2$

$hp(26) + i \cdot hs(26)$ → 1st explore: $(4 + 1 \cdot 2) = 6$ → collision with $hp(17)$

→ 2nd explore: $4 + 2 \times 2 = 8$ → collision with $hp(8)$

→ 3rd explore: $4 + 3 \times 2 = 10$ → collision with $hp(30)$

→ 4th explore: $4 + 4 \times 2 = 12$ → $12 \mod 11 = 1$

→ no collision → put 1

33: $hp(33) = 33 \mod 11 = 0$ → no collision → put 0

14: $hp(14) = 14 \mod 11 = 3$ → no collision → put 3

24: $hp(24) = 2 \mod 11 = 2$ → no no collision → put 2

∴ Therefore, by this process, the final hash table is:

index	0	1	2	3	4	5	6	7	8	9	10
key	33	26	24	14	15	27	17	idle	8	42	30

Dynamic Programming (DP)

当问题同时具备“最优子结构”和“重叠子问题”特性时，DP 是最优解法。

Overview

Simple defintion: Efficient recursion for solving well-behaved optimization problems

Optimization problems: trying to find an optimal solution given some constraints (often discrete problems)

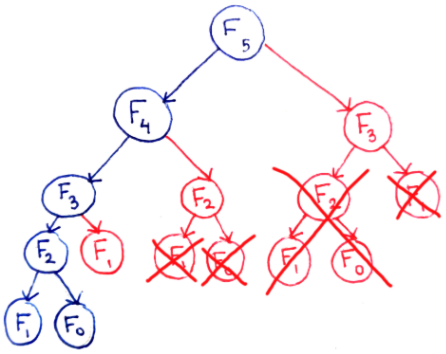
Used for problems with the following properties

- Optimal substructure: optimal solutions incorporate optimal solutions to related subproblems
- Overlapping subproblems: solving the same subproblems over and over again (Memorization exploits this redundancy)

In general, DP good for tasks with small but repetitive search spaces

Examples

Fibonacci series: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. Each value F_i needs to be solved as subproblem for F_{i+1} and F_{i+2}. Similarly, Pascal's triangle.



String/Array algorithms: longest common subsequence, longest increasing subsequence, longest common substring, edit distance, interleaving strings, balanced array, etc.

Graph algorithms: Bellman-Ford (to be discussed later in the course).

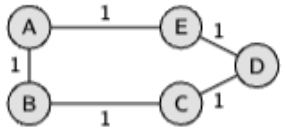
Backpropagation: compute the gradients for one layer at a time starting from the last layer. When calculating gradients for layer i, use the result from layer $i + 1$.

Viterbi algorithm: with a hidden markov model defined by $(S,\ O,\ P(s_{t=0} = s_i),\ P(s_{t+1} = s_j|s_t = s_i),\ P(O_t|s_t = s_i))$, find the most probable state path given the output path. Used for part-of-speech tagging, speech recognition etc. in NLP.

Anti-Examples

Sorting an array of n elements: not an optimization problem.

Longest path problem: find longest path between nodes in a graph without repeating an edge.
This problem does not exhibit optimal substructure.



LongestPath(A,D)=$A \to B \to C \to D$
LongestPath(A,B)=$A \to E \to D \to C \to B$
LongestPath(B,D)=$B \to A \to E \to D$

Compute $n!$ can be done with recursion:

$$\text{fact}(n) = \begin{cases} 1, \text{ if } n = 0 \\ n \cdot \text{fact}(n-1), \text{ if } n > 0 \end{cases}$$

There are no overlapping subproblems, once we compute fact(i), we don't need to compute it again.

UNIVERSITY OF TORONTO

Rod Cutting Problem:

Given: a rod of length n and a table of prices p_i for each rod length $i = 1, ..., n$
Goal: Determine maximum revenue you can obtain from the rod by cutting it into various size pieces and selling them

Visualization:
Suppose we have a rod of length 4 with prices:

length	1	2	3	4
price ($)	1	5	8	9

Possible cuts:
$[1,1,1,1]$: \$4; $[2,2]$: \$10 (optimal);
$[1,3]$: \$9; $[1,1,2]$: \$7; $[4]$: \$9

Naive solution: Try all possible cuts.
This will be $\mathcal{O}(2^n)$, since you either cut or not (2 possibilities for $n-1$ spots)

Rod Cutting Problem with DP

Optimal Substructure:
Suppose in the optimal solution, we make a cut at position k and we have 2 rods
The optimal solution must now contain the optimal way to cut the 2 sub-rods. If not, we could replace them with the optimal sub-solution to obtain a better optimal solution.
$\therefore$ this problem exhibits optimal substructure

Recursive relationship among subproblems:
Let r_n be the maximum revenue you can get from a rod of length n
$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) = \max_{k \in [1,n]} (r_k + r_{n-k})$
We can simplify things. Instead of considering 2 subproblems, we fix the length of the rod on one side. $r_n = \max_{k \in [1,n]} (p_k + r_{n-k})$

Rod Cutting Problem Implementation

Naive implementation:

function Rod-Cut(p, n)
  if $n == 0$ **then**
    Return 0
  $q = -\infty$
  for each $i = 1 : n$ **do**
    $q = \max(q, p[i] + \text{Rod-Cut}(p, n - i))$

However, we can make it more efficient by adding a memory. This gives a top-down approach.

DP with bottom up approach ($\mathcal{O}(n^2)$):

function Rod-Cut(p, n)
  $r[0, ..., n] = 0$
  for each $j = 1 : n$ **do**
    $q = -\infty$
    for each $i = 1 : j$ **do**
      $q = \max(q, p[i] + r[j - i])$
    $r[i] = q$

Example run of Bottom-up approach

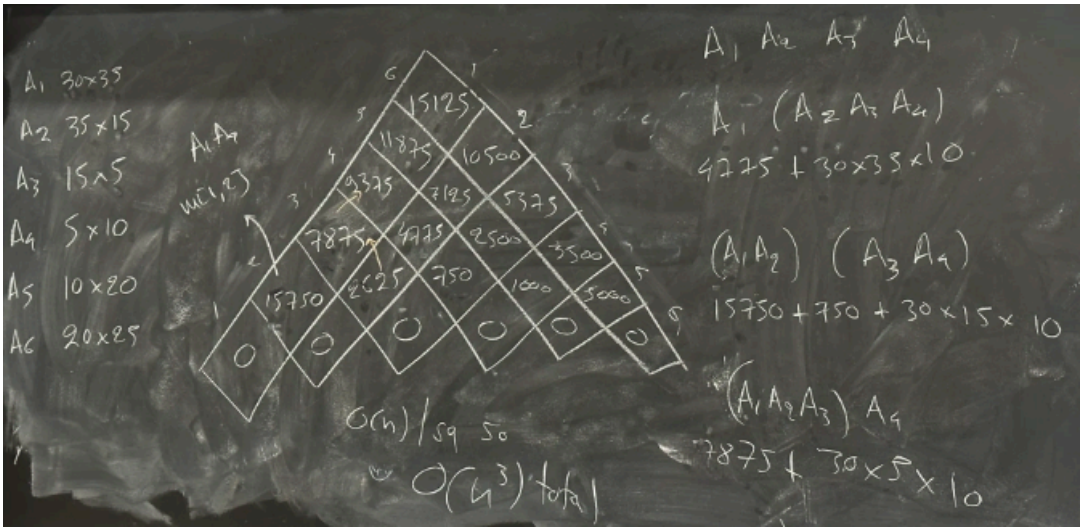
length	1	2	3	4
price ($)	1	5	8	9

p_i=price of rod with length i
r_i=optimal revenue of rod with length i

$j = 1$: $p_1 = 1$, $r_1 = \max(1) = 1$
$j = 2$: $p_2 = 5$, $p_1 + r_1 = 1 + 1 = 2$, $r_2 = \max(5, 2) = 5$
$j = 3$: $p_3 = 8$, $p_1 + r_2 = 1 + 5 = 6$, $p_2 + r_1 = 5 + 1 = 6$, $r_3 = \max(8, 6, 6) = 8$
$j = 4$: $p_4 = 9$, $p_1 + r_3 = 1 + 8 = 9$, $p_2 + r_2 = 5 + 5 = 10$, $p_3 + r_1 = 8 + 1 = 9$,
$r_4 = \max(9, 9, 10, 9) = 10$

Matrix Multiply Problem:

matrix multiplication in lecture:



Greedy Algorithm

Overview

Simplified Definition: an algorithm where locally optimal decisions lead to a globally optimal solution
Idea: when making a choice, take the one that looks the best right now. Local optimality leads to global optimality.
Note: Greedy is not always optimal, but good as approximation algorithms

Properties:

- Greedy choice property: the optimal solution agrees with the first greedy choice
- Smaller subproblems: after making the greedy choice, the resulting subproblem reduces in size
- Optimal substructure: an optimal solution contains optimal solutions to subproblems

A simple example where Greedy is not optimal

Suppose we want to make up x using coins from a given set S of denominations.

Case 1: $S = [1, 2, 4]$. This set is valid, since we can make either even (combinations of 2 and 4) or odd numbers (any even + 1). We can use greedy here, sort the coins from largest to smallest, then always use the largest coin first if not overshooting the target.

Case 2: $S = [1, 4, 5]$. Still valid, since we can make up anything just using \$1. However, greedy won't work here. e.g. if we want \$8, the most optimal way is \$4+\$4. The greedy gives us \$5+\$1+\$1+\$1.

There are only two types of greedy problems you need to consider in this course:

1. Car building: rearrange the full set of tasks to minimize the total penalty
2. Scheduling: find a feasible subset of tasks to maximize the profit

"重排顺序最小惩罚"（Car Building）和 "选取子集最大收益"（Scheduling）。

You can use these two examples as templates for the exams and HWs. The algorithms and proofs should be very similar.

Car building:

> "任务 A 每拖一小时罚 3 块，任务 B 每拖一小时罚 5 块。"

惩罚不是固定金额，而是**按完成时间乘以惩罚系数算的。**

🧱 二、惩罚计算公式（重点）

若我们执行顺序为：

$$J_1, J_2, ..., J_n$$

则每个任务的惩罚 = 它的权重 × 它完成时刻。

$$\text{Total Penalty} = \sum_{i=1}^{n} w_i \times C_i$$

其中：

$$C_i = t_1 + t_2 + ... + t_i$$

就是前面任务的执行时间总和（当前任务结束时刻）。

每个任务的"单位惩罚时间代价"不同，
直觉上：

> 如果某任务需要很长时间（t大），但惩罚又不高（w小），那它性价比低，应该靠后。

所以要最小化：

$$\sum w_i C_i$$

的最优排序是：

$$\boxed{\text{按 } \frac{t_i}{w_i} \text{ 升序排列执行。}}$$

Scheduling (**No Value Version**):

你有一堆任务（或会议、活动），每个都有：

- 开始时间：s_i
- 结束时间：f_i
- 可能还有价值（profit）：v_i

要求从中选出一个**互不重叠**的子集，使得：

- 若任务价值相同 → 选最多的；
- 若任务价值不同 → 总收益最大。

```
def Greedy_Schedule(jobs):
    sort jobs by finish time f_i ascending
    S = []
    last_finish = -∞
    for job in jobs:
        if job.start >= last_finish:
            S.append(job)
            last_finish = job.finish
    return S
```

Scheduling 型问题（调度类贪心）最优解的核心是：

> **"按结束时间早的任务优先"。**
> 也就是说：

$$\boxed{\text{按 } f_i \text{（任务结束时间）升序排序，依次选择不冲突的任务。}}$$

这个策略能保证你选出的任务集合具有**最大数量（或最大收益）**，并且是**全局最优。**

💡 二、贪心思路（为什么按结束时间排序）

假设你当前想排一整天的会议。

直觉上：

> 如果你选了一个很晚才结束的会议，它会挡住后面更多的可能；
> 如果你选一个早结束的，你有更大空间安排后续的会议。

⚙️ 举例

活动	开始 s	结束 f
A	1	4
B	3	5
C	0	6
D	5	7
E	8	9

🔢 步骤 1：按结束时间排序

A(4), B(5), C(6), D(7), E(9)

🔁 步骤 2：扫描选择

初始：空集

- A ✅ (结束于 4)
- B ❌ (3 < 4，有冲突)
- C ❌ (0 < 4，有冲突)
- D ✅ (5 ≥ 4)
- E ✅ (8 ≥ 7)

结果：**{A, D, E}**

🧠 思考：为什么这样选最优?

假设存在一个最优集合 S，它没选最早结束的活动 A，而是选了别的活动 X。
那么由于 A 结束得更早，把 X 替换成 A，不会导致冲突，还可能留出更多空间安排别的活动。
→ 所以，存在另一个不劣的最优解包含 A。

Proof of Correctness: 任务集合 T, 最优解是 J, 贪心算法输出是 G

Notation:
Optimal solution: $J = j_1, j_2, ..., j_n$
Greedy solution: $G = g_1, g_2, ..., g_n$
j_i and g_i are indices into some object set. If the task set $T = \{t_1, t_2, t_3\}$ and $J = j_1, j_2, j_3$, the optimal ordering is $t_{j_1}, t_{j_2}, t_{j_3}$
We treat J and G as ordered sets and use set operations $J \backslash \{j_1\} = j_2, j_3, ..., j_n$, $\{j_0\} \cup J = j_0, j_1, ..., j_n$

Proof of correctness:

1. let $G = g_1, g_2, ..., g_n$
2. First greedy choice: $\exists J = j_1, ..., j_n$ s.t. $g_1 = j_1$
3. Smaller subproblems: After making choice $j_1 = g_1$, we are left with a smaller subproblem
4. Optimal substructure: we must solve the subproblem optimally for the original solution to be optimal. $J \backslash \{j_1\} = j_2, j_3, ..., j_n$ must be an optimal solution to the subproblem
5. Recurse the argument to conclude $j_2 = g_2$, $j_3 = g_3$

$\therefore \exists J$ optimal solution s.t. $J = G$

UNIVERSITY OF TORONTO

🧩 Step 1：定义两个解

- $G = [g_1, g_2, g_3, \ldots]$：贪心算法选的活动集合；
- $J = [j_1, j_2, j_3, \ldots]$：最优解（最多活动数的集合）。

🧩 Step 2：证明贪心的第一个选择是"安全的"

> 贪心算法选的第一个活动 g_1（即最早结束的活动）
> 一定可以出现在某个最优解 J 里。

证明思路：

- 假设最优解 J 的第一个活动是 j_1，
  但它结束得比 g_1 晚（因为贪心选的是最早结束的）。
- 那么你可以把 J 里的 j_1 换成 g_1，
  不会造成任何冲突，还能留出更多时间。
- 所以存在另一个最优解 J'，其中 $j'_1 = g_1$。

✅ **结论：**
g_1 一定在某个最优解中，也就是说它是"安全的第一步"。

🧩 Step 3：形成子问题

既然 g_1 一定没问题，
那我们可以"移除所有与 g_1 冲突的活动"，
得到一个新的子问题：

> "在剩下不冲突的活动里，再选最多个活动。"

这就对应课件那句：

> "After making choice $j_1 = g_1$, we are left with a smaller subproblem."

🧩 Step 4：最优子结构

假设原问题的最优解是 J，

那它包含 g_1 后，剩下部分 $J \setminus \{g_1\}$
也必须是"子问题"的最优解。

换句话说：

> 如果整体最优解包含贪心第一步，
> 那它剩下的部分也必须最优。

否则你可以在剩下部分换成更好的组合，得到更优整体解——矛盾。

🧩 Step 5：递归证明

现在：

- 你知道第 1 步没错；
- 假设前 k 步都选对；
- 第 k+1 步同样可以用同样的逻辑推出；
  → 所以最终每一步都是对的。

结论：

$$\exists J \text{ such that } J = G$$

也就是说——贪心解和最优解完全一致。

🧠 直观理解

- 贪心第一步（选最早结束的活动）不会错；
- 选完它之后，剩下问题还是同样结构；
- 一直递归选下去；
- 每一步都不亏，整体一定最优。

✅ **一句话总结：**

> "贪心算法正确性"的标准证明流程：
> 1. **"第一步选择（greedy choice）不会破坏最优性；"**
> 2. **"剩下问题仍满足最优子结构；"**
> 3. **"递归地重复 → 贪心解 = 最优解。"**

就像"活动选择问题"这样，按结束时间选，
是因为你能用这套五步逻辑**严格证明它不会比最优解差。**

- ✅ **Greedy choice property（贪心选择性质）**
  → "第一步贪心选择可以出现在某个最优解中"。

现在这页是在证明第二个条件：

- ✅ **Optimal substructure（最优子结构）**
  → "拿掉第一步后，剩下的问题仍然必须是最优的"。

只有这两个都成立，贪心算法才整体正确。

Example - easy:

Given: $V = \{v_1, v_2, ..., v_n\}$ set of vehicles, $Q = \{p_1, p_2, ..., p_n\}$ set of corresponding penalties per day vehicle v_i is late.
Currently, all vehicles are late and can only produce one vehicle per day.
Goal: Compute optimal schedule for building vehicles to minimize your total penalty.
Suppose your schedule is $S = s_1, s_2, ..., s_n$, total penalty is $P(S) = \sum_{i=1}^{n} ip_{s_i}$

Devise a greedy algorithm:
Sort Q in descending order, build vehicles from highest penalty to lower. $\mathcal{O}(n \log n)$

```
def BuildCars(penalties):
    sort penalties in descending order
    total_penalty = 0
    for i in range(1, n+1):
        total_penalty += i * penalties[i]
    return total_penalty
```

Prove the greedy choice property:
let $J = j_1, j_2, ..., j_n$ be an optimal solution
Let $G = g_1, g_2, ..., g_n$ be the greedy solution
If $j_1 = g_1$, then done
Suppose $j_1 \neq g_1$
Since we must build all vehicles in the end, g_1 must appear somewhere in J
i.e. $J = j_1, j_2, ..., j_{m-1}, j_m = g_1, ..., j_n$
By definition of greedy choice, $p_{j_1} \leq p_{g_1} = p_{j_m}$ for some $m > 1$
Let J' be a solution where we swap j_1 and j_m, $J' = j_m, j_2, ..., j_1, ..., j_n$

$$P(J) = \sum_{i=1}^{n} i p_{j_i} = 1p_{j_1} + 2p_{j_2} + \cdots + mp_{j_m} + \cdots + np_{j_n}$$

$$P(J') = \sum_{i=1}^{n} i p_{j_m} = 1p_{j_m} + 2p_{j_2} + \cdots + mp_{j_1} + \cdots + np_{j_n}$$

$$P(J') = P(J) - 1p_{j_1} - mp_{j_m} + 1p_{j_m} + mp_{j_1} = P(J) + (m-1)(p_{j_1} - p_{j_m}) \leq P(J)$$

J' is an optimal solution that agrees with the first greedy choice

Prove the problem is reduced to a smaller subproblem after making the first greedy choice:
After scheduling a vehicle on day 1, we have $n-1$ vehicles to schedule in $n-1$ days. Same problem but smaller

How to prove the two props:

证明 Greedy Choice + 证明 Optimal Substructure 的根本意义，就是为了证明——这个贪心算法一定能得到最优解。

🧩 一、Greedy Choice Property（贪心选择性质）

🎯 目标：

证明“贪心算法的第一步选择”一定可以出现在某个最优解中。
👉 也就是“这步是安全的（safe choice）”。

🧠 思路：

用 **交换论证（Exchange Argument）**。

✍️ 通用证明模板：

> 假设存在一个最优解 $J = [j_1, j_2, ..., j_n]$，
> 但贪心算法的第一个选择 $g_1 \neq j_1$。
> 由于 g_1 是贪心算法选出的“最好”的元素，
> 所以有 $\text{value}(g_1) \geq \text{value}(j_1)$。
>
> 于是我们交换 g_1 和 j_1，构造一个新解 J'：
>
> $$J' = [g_1, j_2, ..., j_n]$$
>
> 计算总代价（或总收益）后可得：
>
> $$\text{Cost}(J') \leq \text{Cost}(J)$$
>
> 或
>
> $$\text{Profit}(J') \geq \text{Profit}(J)$$
>
> 所以 J' 仍然是最优的。
> 因此，贪心算法的第一步选择 g_1 一定可以出现在最优解中。

🧩 二、Optimal Substructure（最优子结构性质）

🎯 目标：

证明“做出贪心选择后，剩下的子问题仍是最优的”。

🧠 思路：

用 **反证法（Proof by Contradiction）**。

✍️ 通用证明模板：

> 假设我们已经做出第一步贪心选择 g_1，
> 并得到剩余子问题的最优解 J'。
>
> 假设 J' 不是最优的，
> 那么存在一个更好的子问题解 S，使得：
>
> $$\text{Cost}(S) < \text{Cost}(J')$$
>
> 我们把 g_1 和 S 合并成一个新解：
>
> $$J'' = \{g_1\} \cup S$$
>
> 得到：
>
> $$\text{Cost}(J'') < \text{Cost}(J)$$
>
> （或 Profit 更大）
>
> 这与“J 是最优解”矛盾。
> 因此，J' 必然是子问题的最优解。

Example - Scheduling Deadlines:

Scheduling Deadlines

Given $T = \{t_1, t_2, ..., t_n\}$ a set of tasks, $D = \{d_1, ..., d_n\}$ a set of corresponding deadlines in units of days, $Q = \{p_1, ..., p_n\}$ a set of corresponding profits if we complete the task by the deadline.
Each task takes exactly 1 day to complete. A feasible set is a set of tasks s.t. it is possible to complete all tasks before their deadline
Goal: Find a feasible set of tasks s.t. their profits $\sum_{j \in J} p_j$ are maximized.

e.g.

i	1	2	3	4
t_i	a	b	c	d
d_i	2	1	2	1
p_i	50	10	15	30

Invalid solution: $\{b, d\}$, both needs to completed on day 1
Solution 1: $\{d\}$, profit 30
Solution 2: $\{d, a\}$, profit 30 + 50

Devise a greedy algorithm assuming you have a function Feasible(J) that runs in constant time and check if J is a feasible schedule
Sort tasks in order of decreasing profit. $J = \{\}$, iterate through the tasks, if Feasible($J \cup \{j\}$), then $J = J \cup \{j\}$.

Prove the greedy choice property:
Let $J = j_1, j_2, ..., j_m$ be an optimal solution.
Let $G = g_1, g_2, ..., g_m$ be the greedy solution.
Note that the sequence of selection doesn't matter here. We need to finish all tasks in J and all tasks in G before the corresponding deadlines. If $g_1 \in J$, then done
Assume $g_1 \notin J$, then $p_{g_1} \geq p_j$, $\forall j \in J$ by construction of g_1
Construct $J' = J \backslash \{j\} \cup \{g_1\}$ for any $j \in J$ s.t. J' is feasible.
$\therefore P(J') = P(J) - P(\{j\}) + P(\{g_1\}) = P(J) - p_j + p_{g_1} \geq P(J)$
Thus J' is an optimal solution that agrees with our first greedy choice.

UNIVERSITY OF TORONTO

Prove the problem is reduced to a smaller subproblem after making the first greedy choice:
Start with $T = \{t_1, ..., t_n\}$, make first greedy choice g_1 with profit p_{g_1} where $p_{g_1} \geq p_k, \forall k \in \{1, ..., n\}$
Now, we have $T \backslash \{t_{g_1}\}$ possible tasks remaining, but with less days to complete them
So we have a subproblem T'.
Also, there are possibly some tasks that is now impossible to complete because we don't have enough days and we can remove them. $T' \subset T \backslash \{t_{g_1}\}$
$|T'| \leq |T| - 1 < |T|$, T' is a strictly smaller problem

Prove optimal substructure:
Let $J = j_1, j_2, ..., j_m$ s.t. $g_1 \in J$ be an optimal solution
Let $J' = J \backslash \{g_1\}$ be a solution to the subproblem
Assume that J is optimal and J' is not
Let S be an optimal solution to the subproblem $P(S) > P(J')$ and construct $J'' = \{g_1\} \cup S$. J'' is feasible because we can just do g_1 first and S is feasible for the subproblem.
$P(J'') = P(S) + P(\{g_1\}) > P(J') + P(\{g_1\}) = P(J)$. Contradiction, since J'' is now more optimal.

```
# J = 已选择的任务集合
J = ∅
# Step 1: 按利润从大到小排序任务
Sort tasks by decreasing profit

# Step 2: 遍历任务
for each task j in sorted order:
    if Feasible(J ∪ {j}):    # 加进去后不违反截止时间
        J = J ∪ {j}          # 保留任务
```

Prove problem is reduced to a smaller subproblem after making the first greedy choice 短答

✍️ 三、考试怎么写最简版

一句话就够：

> "After making the first greedy choice g_1, the remaining tasks form a smaller instance of the same problem (same constraints, same objective), thus we can recursively apply the greedy rule."

或者更口语一点：

> "Once we fix the first greedy choice, what's left is the same scheduling problem but smaller."

在证明 **Greedy choice property** 时，
你只需要证明 ——g_1（第一次贪心选择）可以出现在某个最优解中，
而 不要求它一定在最优解的第一位。

2024

1. **[Asymptotics, Recurrences, Induction, Combinatorics]**
   Short Answers, 15+10+10+15 points

   (a) Firstly, we show $\log(n!) = \Omega(n \log n)$.

$$\begin{aligned}\log(n!) &= \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lfloor n/2 \rfloor}^{n} \log i \geq \sum_{i=\lfloor n/2 \rfloor}^{n} \log \frac{n}{2} = \left\lceil \frac{n}{2} \right\rceil \log \frac{n}{2} \geq \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2} \geq \frac{1}{4} n \log n\end{aligned}$$

   Then we show $\log(n!) = \mathcal{O}(n \log n)$ in a similar way. Note that $\log i \leq \log n$ for $i \leq n$

$$\begin{aligned}\log(n!) &= \sum_{i=1}^{n} \log i \\ &= \log 1 + \log 2 + \cdots + \log n \\ &\leq \log n + \log n + \cdots + \log n \\ &\leq n \log n\end{aligned}$$

   (b) $a = 4$, $b = 16$, $f(n) = n \log n$.
   $\log_b a = \log_{16} 4 = \frac{1}{2}$, and $f(n) = n \log n = \Omega(n^{\frac{1}{2}+\epsilon})$ for $\epsilon \in (0, \frac{1}{2})$.
   Case 3. Check the regularity condition:
   $af(\frac{n}{b}) = 4(\frac{n}{16} \log \frac{n}{16}) = \frac{n}{4}(\log n - \log 16) = \frac{n}{4} \log n - n \leq c(n \log n)$ for $c \in [\frac{1}{4}, 1]$.
   Thus the regularity condition is satisfied, and $T(n) = \Theta(n \log n)$

   (c) For all integers $n \geq 0$, let $P(n)$ be the proposition that $\sum_{i=0}^{n} i \cdot i! = (n+1)! - 1$.
   Inductive basis: $P(0)$ is true since $\sum_{i=0}^{n} i \cdot i! = 0 \cdot 0! = 0 = 1! - 1 = (0+1)! - 1$.
   Inductive hypothesis: Assume $P(n)$ is true, that is, $\sum_{i=0}^{n} i \cdot i! = (n+1)! - 1$.
   Inductive step: Consider $P(n+1)$.

$$\begin{aligned}\sum_{i=0}^{n+1} i \cdot i! &= (n+1)(n+1)! + \sum_{i=0}^{n} i \cdot i! \\ &= (n+1)(n+1)! + (n+1)! - 1 \qquad \text{(by the induction hypothesis)} \\ &= (n+2)! - 1\end{aligned}$$

   Therefore, $P(n+1)$ is true.

   Since $P(0)$ is true, and $P(n)$ is true implies $P(n+1)$ is true, we have shown through mathematical induction that $P(n)$ is true for all $n \geq 0$.

   (d) **Combinatorial Question**: How many ways can we select a subgroup of size k from a group of size n, with 2 leaders?
   Left-Hand Side: We have n people to select from. Firstly, choose 2 people as the leaders. There are $\binom{n}{2}$ ways. Then we select the rest $k-2$ people to form the subgroup from the remaining $n-2$ people. There are $\binom{n-2}{k-2}$ ways. By rule of product, there are $\binom{n}{2}\binom{n-2}{k-2}$ ways to form the subgroup.

Right-Hand Side: We firstly select k people from n to form the subgroup. There are $\binom{n}{k}$ ways. Then within the subgroup, select 2 leaders. There are $\binom{k}{2}$ ways. Therefore, there are $\binom{n}{k}\binom{k}{2}$ ways to form the subgroup.

2. **Algorithm Design, 25 points**

Algorithm: Build a min-heap H of size k to keep track of the smallest elements from each of the k arrays. Repeatedly extract the smallest element (s, i) from the heap (i is the index of the array from which s originates). Compare this element with previously extracted element x (if any). If $s = x$, return True. Otherwise, insert the next element from A_i. If no duplicate is found after all N elements are visited, return False.

Proof of correctness: Since all the k lists are sorted, we know that the values visited (extracted from heap) must be no larger than any unvisited elements in the original array. In each iteration, we use the min-heap property to always select the minimum value in the currently pointed elements $v_1, ..., v_k$. Then the minimum value x must be at least as large as the previously extracted element s, but smaller than any unvisited elements or elements in the heap. If there are any duplicates in the k arrays, one of them will be saved as s, and the other one will be x just extracted from the heap. Hence, no duplicates will be missed. In case there are no duplicates, the algorithm terminates after visiting all N elements.

Running time: It takes $\mathcal{O}(k)$ to build a heap of size k on the initial k elements from the k arrays. Both extract min and insertion take $\mathcal{O}(\log k)$, and we need to repeatedly extract min and insert the remaining $N-k$ elements to check through all N elements in the worst case. At the end of this procedure, there will still be k elements remaining in the heap. In order to check for duplicates in the remaining k, we need to extract those as well. This will take $(N-k+k)\mathcal{O}(\log k) = N\mathcal{O}(\log k)$. So total runtime is $\mathcal{O}(k) + N\mathcal{O}(\log k) = \mathcal{O}(N \log k)$, since $N > k$.

Space complexity: $\mathcal{O}(k)$ as the heap of size k is built out-of-place.

3. **Dynamic Programming, 15+10 points**

(a) The hedgehog cannot move up or to the left, so it can not revisit any grid it has already visited. For each grid with coordinates i, j, the maximum number of apples the hedgehog can collect up to this grid depends on the maximum apples collected from the grid above it(coordinates $i-1, j$) and the grid to its left (coordinates $i, j-1$). From this, a base case naturally follows: the only grid that cannot be reached from above or from the left is the initial top-left grid. As an edge case, when either the row or the column has index 0, the maximum number of apples depends only on one grid instead of two (since the forest ends at the boundaries of the area and no apples can be collected there). These give a recurrence relation similar to the longest common subsequence problem:

$$dp[i,j] = \begin{cases} \max(dp[i-1,j], dp[i,j-1]) + apples[i,j] & 1 \le i < n \text{ and } 1 \le j < m \\ dp[i-1,j] + apples[i,j] & j = 0 \\ dp[i,j-1] + apples[i,j] & i = 0 \\ apples[0,0] & i, j = 0 \end{cases}$$

(b) This can be implemented in a bottom-up manner.

Bottom-Up Solution:

```
1:  procedure HEDGEHOG-IN-FOREST-BOTTOM-UP(apples, n, m)
2:      initialize dp[0..n-1][0..m-1]
3:      dp[0][0] = apples[0][0]                          ▷ last case of recursive relation
4:      for i = 1 to n-1 do
5:          dp[i][0] = dp[i-1][0] + apples[i][0]         ▷ second case of recursive relation
6:      for j = 1 to m-1 do
7:          dp[0][j] = dp[0][j-1] + apples[0][j]         ▷ third case of recursive relation
8:      for i = 1 to n-1 do
9:          for j = 1 to m-1 do
10:             dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + apples[i][j]
11:     return dp[n-1][m-1]
```

The runtime of this algorithm is $\mathcal{O}(nm)$, since a nested for-loop on lines 8-10 will be the dominating factor in the total runtime, and all operations inside the inner loop take $\mathcal{O}(1)$ time. Space complexity is $\mathcal{O}(nm)$ due to the DP lookup table.

2021

1. **[Asymptotics, Recurrences, Combinatorics, Induction]**
   Short Answers, 15+10+10+15 points

   (a) $\lg n \ll n^2 \ll n^{\lg n} \ll 2^n \ll 2^{2n}$

   - All poly-logarithmic functions are upper-bounded by all polynomial functions $\implies \lg n \ll n$
   - $2 \ll \lg n \implies n^2 \ll n^{\lg n}$
   - $n^{\lg n} = 2^{\lg(n^{\lg n})} = 2^{\lg n \cdot \lg n} = 2^{(\lg n)^2}$ and $(\lg n)^2 \ll n \implies n^{\lg n} \ll 2^n$
   - $\lg\left(\lim\limits_{n\to\infty} \dfrac{2^n}{2^{2n}}\right) = \lim\limits_{n\to\infty} \left(\lg \dfrac{2^n}{2^{2n}}\right) = \lim\limits_{n\to\infty} (n \cdot \lg 2 - 2n \cdot \lg 2) = \lim\limits_{n\to\infty} (n - 2n) = \lim\limits_{n\to\infty} -n = -\infty$

     Therefore, $\lim\limits_{n\to\infty} \dfrac{2^n}{2^{2n}} = 2^{-\infty} = 0$

   (b) (i) Since $\log_{27} 3 = 1/3$, letting $0 < \epsilon < 1/6$ gives Case 3 of the Master Theorem.

   Checking the regularity condition: $3\sqrt{\dfrac{n}{27}} = \left(\dfrac{1}{\sqrt{3}}\right)\sqrt{n}$. Thus, regularity is satisfied if $\dfrac{1}{\sqrt{3}} \le c < 1$.

   Therefore we have, $T(n) = \Theta(\sqrt{n}) = \Theta(n^{0.5})$.

   (ii) Since $\log_4 8 = 1.5$, letting $0 < \epsilon < 0.5$ gives Case 1 of the Master Theorem. Therefore we have, $T(n) = \Theta(n^{\log_4 8}) = \Theta(n^{1.5}) = \Theta(n\sqrt{n})$.

   (c) **Question**: Suppose we are creating a random string of n characters. The characters are selected from a pool containing x total letters and y total numbers. How many random strings of length n can we create (the string can contain repeated characters)?

   Left-Hand Side: For each position in the string, the characters has $x + y$ possible values. Therefore, there are $(x + y)^n$ possible strings.

   Right-Hand Side: If the string contains k letters, then it must contain $n - k$ numbers. Thus, there are x^k ways to pick the k letters and y^{n-k} ways to pick the numbers, and $\binom{n}{k}$ ways to permute them. Finally, we sum all the possible values of k to get all possible strings.

   (d) **Base case**: $n = 1$, $2^{2\cdot 1} - 1 = 2^2 - 1 = 4 - 1 = 3$, which is divisible by 3.

   Hypothesis: $2^{2n} - 1$ is divisible by 3, i.e. $2^{2n} - 1 = 3m$ for some positive integer m.

   Induction Step: Consider $2^{2(n+1)} - 1$

   $$2^{2(n+1)} - 1 = 2^{2n+2} - 1 = 4(2^{2n}) - 1 = 4(3m + 1) - 1 = 12m + 4 - 1 = 12m + 3 = 3(4m + 1)$$

   Therefore, $2^{2(n+1)} - 1$ is divisible by 3

2. **Algorithm Design, 15+10 points**

(a) **Algorithm**: To find a and b, we first insert the largest element of each array into a max-heap. While inserting keep track of the smallest element placed in the max-heap (the minimum of the maximum elements).

Obtain the maximum element in the heap (at the root) and the minimum element in the heap (found in the previous step). Calculate and store the difference between these two elements in the variable `range`.

Extract the maximum from the heap, and replace it with the next largest element in the list it came from (then re-heapify). If necessary, update the stored value for the smallest element in the heap and the `range`.

If the new `range` is smaller than any previous range, store that value in `minimum_range` along with the value of the corresponding elements. (i.e. `minimum_range` = min(`range`, `minimum_range`))

Repeat the above two steps until one of the arrays is empty. Return `minimum_range` and the corresponding elements that produces this range.

Runtime: This algorithm requires building a k-element max-heap, as well as removing/inserting $n \cdot k$ elements from/into that heap. Therefore the overall runtime is $O(k + n \cdot k \cdot \lg k) = O(n \cdot k \cdot \lg k)$.

Correctness: This algorithm can be incorrect in two possible ways. *(i)*, a, and b do not satisfy that the range between them contains at least one element from every list, or *(ii)*, $b - a$ is not minimized.

(i): Any possible selection for a and b are at some point the minimum and maximum elements in a max-heap containing one element from each array. Therefore, it must be the case that there is at least one element, c, in each array such that $a \leq c \leq b$, for any possible a, b under consideration. Therefore the algorithm will not be incorrect in this way.

(ii): If $b-a$ is not minimal, there must be some b^*, a^* such that $b^* - a^*$ is minimal. *ATaC* that $b^* - a^*$ is minimal, but our algorithm did not choose them. Consider that because we only remove one element at a time, and because b^* is greater than at least one element in each of the other lists, b^* must have been the root of our max heap at some point over the course of the algorithm. Given that we did not select a^*, b^*, it must be that a^* was never the minimum element in the heap when b^* was the root. However we know that at least one element from each list falls between a^* and b^*, and those elements would not have been removed from the heap before b^*, therefore if a^* was present in the heap with b^*, it must have been the minimum element. Therefore a^* must have never been in the heap with b^*. However, since we know that every other list contains some element between a^* and b^*, and whatever element from the list of a^* that was on the heap with b^* is greater than a^*, then whatever was the minimum element on the heap at this time would be greater than a^* while still, together with b^*, forming a range containing at least one element from each list. However, this contradicts our assumption that $b^* - a^*$ is minimized.

(b) The most efficient way to perform this algorithm is the same as above, however the lists must first be sorted. This takes $O(k \cdot n \lg n)$ time using heapsort. Therefore, in total, the algorithm will take $O(n \cdot k \cdot \lg k + k \cdot n \lg n)$. We did not specify that $k < n$, so this should not be simplified further.

3. **Dynamic Programming, 10+15 points**

(a) On the string `AAAABA`, the greedy algorithm would produce `AAAA|B|A`, but the optimal solution is `AAA|ABA`.

(b) The key idea is to isolate the last palindrome in the string, check all possible last palindromes, and then use the optimal substructure to obtain the number of cuts required for the rest of the string.

Let $C[j]$ be the minimum number of cuts required to partition the substring $s_{1,j}$ into palindromes.

$$C[j] = \begin{cases} 0 & \text{if } s_{0,j} \text{ is a palindrome} \\ \min\limits_{\substack{0 \le k < j \\ s_{k+1,j} \text{ is a palindrome}}} \Big(C[j],\ 1 + C[k] \Big) & \text{if } s_{0,j} \text{ is not palindrome} \end{cases}$$

Bottom-Up Solution:

```
1:  procedure MINIMUM-PALINDROMIC-CUTS-BOTTOM-UP(s)
2:      n ← LENGTH(s)
3:      initialize C[1..n]
4:      for j = 1 to n do                          ▷ initialization for comparison later
5:          C[j] ← n                               ▷ n upper bounds the total number of possible cuts
6:      for j = 1 to n − 1 do
7:          if IS-PALINDROME(s_{1,j}) then         ▷ s_{1,j} is a palindrome, therefore no cuts
8:              C[j] ← 0
9:          else                                   ▷ s_{1,j} is not a palindrome
10:             for k = 1 to j − 1 do              ▷ we need to make some cut
11:                 if IS-PALINDROME(s_{k+1,j}) then   ▷ s_{i+1,j} is a palindrome
12:                     C[j] ← min(C[j], 1 + C[k])     ▷ make cut s_{1,k}|s_{k+1,j}
13:     return C[n]
```

This algorithm has a total running time of $\mathcal{O}(n^2)$, since it requires 2 nested for loops.

The space complexity is $\mathcal{O}(n)$, since we need to keep track of a string of length n and an array $C[1..n]$.

Top-Down with Memoization Solution:

Note: Assume when the function is called, it is initially called with $C[1..n]$ with all elements equal to n.

```
1:  procedure MINIMUM-PALINDROMIC-CUTS-TOP-DOWN(s, n, C)
2:      if C[n] < n then return C[n]      ▷ This means we have already calculated this sub-problem
3:      if IS-PALINDROME(s_{1,n}) then    ▷ s_{1,n} is a palindrome, therefore no cuts
4:          C[n] = 0
5:          return C[n]
6:      for k = 1 to n − 1 do             ▷ we need to make some cut
7:          if IS-PALINDROME(s_{k+1,n}) then   ▷ s_{i+1,n} is a palindrome
8:              C_k = MINIMUM-PALINDROMIC-CUTS-TOP-DOWN(s, k, C)
9:              C[n] ← min(C[n], 1 + C_k)      ▷ make cut s_{1,k}|s_{k+1,n}
10:     return C[n]
```

This algorithm's running time is described by the recurrence $T(n) = \left(\sum_{k=1}^{n-1} T(k)\right) + \mathcal{O}(1)$. However, since Memoization is used, the value of $T(k)$ may have already been computed, making that call $\mathcal{O}(1)$. We notice there are n distinct inputs to the recurrence (all of the digits $1, \ldots, n$). Therefore, in exactly n of the calls to the recurrence we will have to compute to answer, and in the rest the answer is stored. Suppose all sub-problems have been pre-computed. Therefore, $T(n) = \left(\sum_{k=1}^{n-1} \mathcal{O}(1)\right) + \mathcal{O}(1) = \mathcal{O}(n)$. Now, we multiply by n to account for the n times where we do not have the answer stored (Since in the worst case each $T(k) = \mathcal{O}(n)$). Therefore, the $\boxed{\text{total running time is } \mathcal{O}(n^2)}$.

The $\boxed{\text{space complexity is } \mathcal{O}(n)}$ since we need to keep track of a string of length n and an array $C[1..n]$.