Computational Tractability

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

$2n^2$ for stable matching with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants c > 0 and d > 0 such that on every input of size N, its running time is bounded by $c N^d$ steps.

$c(2N)^d = c2^dN^d$

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst-case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst-Case Polynomial

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

n! for stable matching with n men and n women $\longrightarrow$ $O(n^2)$

Exceptions.
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method
Unix grep

In mathematical optimization, Dantzig's **simplex algorithm** (or **simplex method**) is a popular algorithm for linear programming.

https://en.wikipedia.org/wiki/Simplex_algorithm

Asymptotic Order of Growth

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Slight abuse of notation. $T(n) = O(f(n))$.
- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.
- Statement doesn't "type-check."
- Use Ω for lower bounds.

Properties

Transitivity.
- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.
- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$.

Some common bounds

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

can avoid specifying the base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

every exponential grows faster than every polynomial

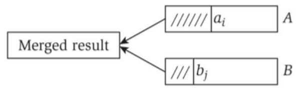
Linear Time: O(n)

Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else         append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.
Pf. After each comparison, the length of output list increases by 1.

O(nlogn) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

sorting takes O(nlogn)

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

take O(n)

Quadratic Time : $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time : $O(n^3)$

$O(n^4)$?

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time : $O(n^k)$

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S is an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \le \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

poly-time for k=17, but not practical

Expotential Time (k^n)

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

2^n: total number of subsets of an *n*-element set

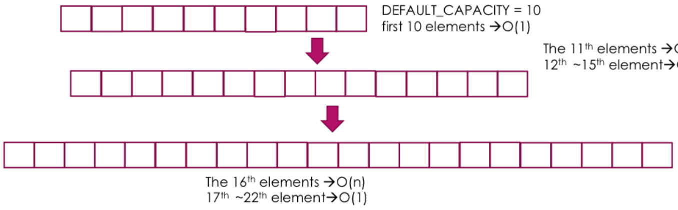
n^2

```
S* ← φ
foreach subset S of nodes {
   check whether S is an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

实验课

Amortized Analysis 摊还分析

Aggregate method 聚集法



Obviously， worst case operation cost O(n), but we shouldn't consider that insert n elements will cost $O(n^2)$

- $T(n) = \sum_{i=1}^{n} c_i$

$$= \sum_{i=1, i-1\neq 10\times 1.5^k}^{n} c_i + \sum_{i=1, i-1=10\times 1.5^k}^{n} c_i$$

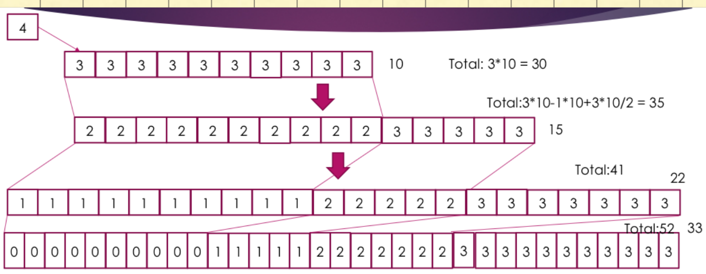
$$= \sum_{i=1, i-1\neq 10\times 1.5^k}^{n} 1 + \sum_{i=1, i-1=10\times 1.5^k}^{n} (10\times 1.5^k + 1)$$

$$= n + \sum_{k=0}^{\log_{1.5}\frac{n}{10}} 10\times 1.5^k = n + 10 * \left(\frac{1-1.5^{\log_{1.5}\frac{n}{10}+1}}{1-1.5}\right) = n + 10 * \left(\frac{1-1.5*\frac{n}{10}}{-0.5}\right)$$

$$= n + 3n - 20 = 4n - 20$$

- $\frac{T(n)}{n} = O(1)$
- Pushing an element onto the dynamic array takes constant time when the worst case operation's cost amortize to other operations.

Accounting method 会计法



Potential method 势能法

function Φ

Φ : maps states of the data structure to non-negative numbers.

$$\boxed{T_{\text{amortized}}(o) = T_{\text{actual}}(o) + C\cdot(\Phi(S_{\text{after}}) - \Phi(S_{\text{before}})),}$$

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. constant C is usually omitted when using big O notation.

For any sequence of operations $O = o_1, o_2, \ldots, o_n$, define:

- The total amortized time: $T_{\text{amortized}}(O) = \sum_{i=0}^{n} T_{\text{amortized}}(o_i)$,
- The total actual time: $T_{\text{actual}}(O) = \sum_{i=0}^{n} T_{\text{actual}}(o_i)$.

Then:

$$T_{\text{amortized}}(O) = \sum_{i=1}^{n} (T_{\text{actual}}(o_i) + C\cdot(\Phi(S_i) - \Phi(S_{i-1}))) = T_{\text{actual}}(O) + C\cdot(\Phi(S_n) - \Phi(S_0)),$$

$$T_{\text{actual}}(O) = T_{\text{amortized}}(O) - C\cdot(\Phi(S_n) - \Phi(S_0)).$$

Since $\Phi(S_0) = 0$ and $\Phi(S_n) \geq 0$, $T_{\text{actual}}(O) \leq T_{\text{amortized}}(O)$

so the amortized time can be used to provide an accurate upper bound on the actual time of a sequence of operations, even though the amortized time for an individual operation may vary widely from its actual time.

- Define Φ = 3(*n*-*1*) - 2*N, n means the number of the elements, N means the capacity.*
- S_0 *n =11 N=15(ArrayList is special, it has capacity 10 when there is an element. So, we do our analysis with N = 15 the Arraylist initialized with having11 elements and capacity 15)*
- *when n = N*2/3+1* , Φ >= 0; when n = 11, N = 15, Φ = 0; when n=N, Φ should be >=n to pay for the enlarging the table; When insert a element, but not enlarge the table

$$\boxed{\begin{aligned} T_{amortized}(o_i) &= T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1}) \\ &= 1 + (3*(n_i - 1) - 2N_i) - (3*(n_{i-1} - 1) - 2N_{i-1}) = 1 + 3*n_i - 3*n_{i-1} = 4 \end{aligned}}$$

- When insert a element, and enlarge the table

$$\boxed{\begin{aligned} T_{amortized}(o_i) &= T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1}) \\ &= i + (3*(n_{i-1} + 1 - 1) - 2*1.5*N_{i-1}) - (3*(n_{i-1} - 1) - 2N_{i-1}) = i + (3 - N_{i-1}) = 4 \end{aligned}}$$

$$\sum_{i=1}^{n} T_{amortized}(o_i) = 4n$$

- this shows that any sequence of *n* dynamic array operations takes O(*n*) actual time in the worst case.