Fundamentals of algorithm and asymptotic analysis

① algorithms

1) what is an algorithm?

↳ **computational problem**: a (possibly infinite) set of (x, S) pairs, where x is a valid input and S is a set of valid solutions for x

↳ an **algorithm** provides the "how" — a well defined procedure that outputs a valid solution, given some input x

2) what do we want in an algorithm?

↳ **correctness**: an algorithm is said to be correct iff for every valid input, it halts w/ a valid solution. This is a "worst case" notion of correctness, that it is always correct

↳ **running time**: the point of running time analysis is to determine how the number of steps executed by an algorithm scales w/ input size

3) notions of input size

array	→ number of array elements	always important to define
integer	→ number of bits in binary	
graph	→ $\|V\| + \|E\|$	

---

② models of computation (how we define a step)

comparison model

Definition 1.4.1. *In the* comparison model, *the input is an array A of n numbers, and an algorithm can compare two elements ("Is $x > y$, $x < y$, or $x = y$?") in one time unit. No other[4] operations on the elements are allowed. The array can be manipulated (e.g., permuted or broken into two sub-arrays) without operating on the elements at no cost.*

query model

Definition 1.4.9. *In the* string query model, *the input is a string of n bits. In one time unit, an algorithm can query one bit of the string. All other operations on the string can be performed at no cost.*

The total number of bits of the input string queried by an algorithm is called its *query complexity*.

Definition 1.4.11. *In the* graph query model, *the input is the symmetric adjacency matrix G of an n-node undirected graph. In one time unit, an algorithm can query one entry in the matrix G. All other operations on G are free.*

word RAM model

The framework we mostly use subsequently is a conventional **word RAM** model, where the 'programming language' consists of sequential instructions and the running time is the total number of instructions executed. What constitutes an 'instruction'? We omit a formal specification, but it consists of the typical basic operations found in standard programming languages: arithmetic operations ($+, -, \times, /,$ mod $, <, >, \lfloor\cdot\rfloor, \ldots$), memory access operations, and control flow operations (`if, return, ...`). Each instruction is allowed to operate on a *word* of data, and the word size is limited. (Clearly, it would be useless to have a model where each instruction could operate on inputs of unrestricted size in one time unit.) We typically assume that numbers (e.g., entries in an array) are each of word-size so that they can be operated on in unit time.

---

(3) analysis of lower bounds — to be rigorous, we should also argue for the existence of the algorithm that runs in N^* steps, vs $< N^* \Rightarrow$ min is N^*.

Adversary arguments used to prove lower bounds for the running time in concrete computation models. The adversary, by adjusting his responses, makes sure that if an algorithm takes too few steps, then it can come up w/ at least two different solutions that cannot be differentiated by the algorithm.

↓ input decided on-the-fly by adversary who keeps options open about what actual input is. Adversary ensures that if algorithm takes too few steps, then there are at least two valid inputs which are consistent w/ the results of the algorithm's steps, yet the solutions for the two inputs are different.

(eg)

Claim 1.4.3. *Any algorithm solving* Max *must have running time* $\geq n-1$ *in the comparison model.*

Proof. Fix any algorithm $\mathcal{M}$ that correctly solves Max on all inputs. The idea of the proof is to show that if $\mathcal{M}$ always makes less than $n-1$ comparisons, then there are two arrays A and A' so that the two cannot be differentiated based on $\mathcal{M}$'s comparisons. However, the maximum elements of A and A' occur at different locations, and so, $\mathcal{M}$ must err on either A or A', a contradiction.

(1) Take an input A for which $\mathcal{M}$ makes $< n-1$ comparisons. Construct a graph G on n nodes (indexed by $1, \ldots, n$), where nodes i and j are adjacent iff $\mathcal{M}$ compares A_i and A_j. Since G has $< n-1$ edges, it is disconnected. That is, there exists a partition of the nodes into C_1 and C_2 such that for any $i \in C_1, j \in C_2$, there is no edge between i and j, and so, A_i and A_j are not compared by $\mathcal{M}$.

(2) Suppose given A as input, $\mathcal{M}$ outputs A_{i^*} as the maximum. Without loss of generality, let $i^* \in C_1$. (3) Now, consider a new array A', such that if $i \in C_1$, $A'_i = A_i$, but if $i \in C_2$, $A'_i = A_i + m$ where m is sufficiently large that the maximum element of A' is in C_2. See an example in Figure 1.1.

(4) Observe that the comparisons made by $\mathcal{M}$ cannot distinguish between A and A', as the only comparisons which are different are between $i \in C_1$ and $j \in C_2$, and $\mathcal{M}$ does not make such comparisons. So, $\mathcal{M}$ must give a wrong answer for either A or A'.[5] □

(A)

1. Take an input
2. Describe some model with which algo M operates on input w/ $< N^*$ operations, and describe its conclusion
3. Modify input such that solution is different + proof that solutions different
4. Show that M gives same solution for both inputs

(B)

1. Describe algo, and describe how adversary replies
2. Show that w/ $< N^*$ steps, there are at least two different inputs that are consistent w/ algo
3. Show that inputs (and their solutions) are diff, so one must be wrong

In the context of the graph query model, a problem is evasive if it requires $\binom{n}{2}$ queries.

Claim 1.4.12. *The problem of determining whether a graph is connected is evasive.*

Proof. We again use an adversary argument. Let $\mathcal{M}$ be an algorithm making $m < \binom{n}{2}$ queries which decides whether its input is connected or not. (1) It issues a sequence of queries $(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$, and the adversary replies with $b_1, b_2, \ldots, b_m$.

We now define how the adversary replies to each query. Suppose the adversary receives the ℓ'th query for $1 \leq \ell \leq m$. The adversary's strategy for choosing b_ℓ is as follows. She considers the graph $G^{(\ell)}$ defined by: $G^{(\ell)}[i_1, j_1] = b_1, \ldots, G^{(\ell)}[i_{\ell-1}, j_{\ell-1}] = b_{\ell-1}$, $G^{(\ell)}[i_\ell, j_\ell] = 0$ and all other entries equal to 1. That is, $G^{(\ell)}$ is the graph which is consistent with the previous query replies, does not have the edge (i_ℓ, j_ℓ) and has all other unqueried edges. If $G^{(\ell)}$ is connected, she replies $b_\ell = 0$; otherwise, she replies $b_\ell = 1$.

(2) Now, after $\mathcal{M}$ makes all its m queries, we can define two graphs. The first graph, which we call G_0, is consistent with the adversary's replies but sets all the unqueried edges to 0. The second graph, which we call G_1, is also consistent with the adversary's replies but sets all the unqueried edges to 1. $\mathcal{M}$ cannot (3) distinguish between G_0 and G_1 because both are consistent with its queries. We argue below that G_0 is disconnected while G_1 is connected, and hence, $\mathcal{M}$ must err on one of them, finishing the proof.

Clearly, G_1 is connected by the definition of the adversary's strategy.

Lemma 1.4.13. *G_0 is disconnected.*

Proof. Consider a pair of nodes i and j such that (i, j) was not queried by $\mathcal{M}$. We claim that there is no path between i and j in G_0.

Let's argue by contradiction. Suppose there is a path between i and j in G_0. This path only consists of queried edges to which the adversary replied 1. Let (i', j') be the edge on this path that was queried last by $\mathcal{M}$. Now, observe that according to the adversary's strategy given above, there is a path from i' to j' in the graph G_{con} she considers at this stage. This is because in G_{con}, the path from i' to i, the path from j to j', and the edge from i to j are all present. Thus, she must reply 0 to the query (i', j'), a contradiction! □

Decision trees

↳ node : comparison
↳ branch : outcome of comparison
↳ leaf : output / solution

} we can use a decision tree to model any comparison based algorithm. worst case running time = height

1. A height-h binary tree has $\leq 2^h$ leaves.
2. A height-h binary tree has $\leq 2^{h+1} - 1$ nodes

$h = \lfloor \lg \text{leaves} \rfloor + 1$

no. of nodes $= \underbrace{2^0 + 2^1 + 2^2 \ldots}_{h}$

GP: $\frac{a(r^h - 1)}{r - 1} = \frac{(1)(2^{h+1} - 1)}{(2 - 1)}$

Theorem: Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Claim 1: A height h binary tree has $\leq 2^h$ leaves
Claim 2: The decision tree must contain $n!$ leaves

Recall: Worst-case running time is the **height** of the decision tree.

Ask ourselves: We have $n!$ leaves. What's our *minimum* height? How to relate n and h? It's $n! \leq 2^h$

$$\begin{aligned} h &\geq \lg(n!) \text{ (lg is monotonically increasing)} \\ &\geq \lg((n/e)^n) \quad \text{(Stirling's formula)} \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \end{aligned}$$ □

④ asymptotic analysis

1) asymptotic bounds

idea: we can describe how our algorithm's steps scale in input size by using functions to describe their bounds and shape. In particular, we use asymptotic bounds — i.e. as input scales past a certain point, can we bound $f(n)$ by some scaled function $c \cdot g(n)$?

Big O $\exists\, c > 0,\ n_0 > 0$ s.t. $\forall n > n_0,\ 0 \le f(n) \le c \cdot g(n)$. $f(n) = O(g(n))$

↳ there is at least one (c, n_0) pair where $f(n) \le c \cdot g(n)$ after n_0

i.e. there is at least one way you can tweak $g(n)$ s.t. $g(n)$ overtakes $f(n)$ at some point

---

small o $\forall c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0,\ 0 \le f(n) < c \cdot g(n)$. $f(n) = o(g(n))$

↳ for all c, there is a point after which $f(n)$ is strictly bound by $c \cdot g(n)$

ie. no matter how you tweak the slope, $g(n)$ will overtake $f(n)$ @ some point

---

Big omega $\exists\, c > 0,\ n_0 > 0$ s.t. $\forall n > n_0,\ f(n) \ge c \cdot g(n) \ge 0$. $f(n) = \Omega(g(n))$

↳ there is at least on (c, n_0) pair where $f(n) \ge c \cdot g(n)$ after n_0

i.e. there is at least one way you can tweak $g(n)$ so that $f(n)$ overtakes $g(n)$ at some point

---

small omega $\forall c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0,\ f(n) > c \cdot g(n) \ge 0$. $f(n) = \omega(g(n))$

↳ for all c, there is a point after which $f(n)$ is strictly greater than $c \cdot g(n)$

i.e. no matter how you tweak $g(n)$, $f(n)$ will overtake $g(n)$ at some point

---

theta $f(n) = \Theta(g(n))$ iff. $f(n) = o(g(n))$ and $f(n) = \Omega(g(n))$

2) asymptotic bounds as sets

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

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

$$\begin{aligned}\Theta(g(n)) &= \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \le c_1 \cdot g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} \\ &= O(g(n)) \cap \Omega(g(n))\end{aligned}$$

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } 0 \le f(n) < c \cdot g(n) \text{ for all } n \ge n_0\}$$

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } 0 \le c \cdot g(n) < f(n) \text{ for all } n \ge n_0\}$$

↳ observe that $f(n)$ cannot simultaneously be $o(g(n))$ and $\omega(g(n))$

i.e. small − big cannot occur simultaneously, nor small − small.

3) properties of asymptotic bounds

l'hospital's rule

$$\lim_{x\to a}\frac{f(x)}{g(x)}=\frac{0}{0} \quad \text{OR} \quad \lim_{x\to a}\frac{f(x)}{g(x)}=\frac{\pm\infty}{\pm\infty}$$

$$\lim_{x\to a}\frac{f(x)}{g(x)}=\lim_{x\to a}\frac{f'(x)}{g'(x)}$$

limits → provide different "views" of f(n)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0 \implies f(n)=o(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}<\infty \implies f(n)=O(g(n))$$

$$0<\lim_{n\to\infty}\frac{f(n)}{g(n)}<\infty \implies f(n)=\Theta(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}>0 \implies f(n)=\Omega(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty \implies f(n)=\omega(g(n))$$

reflexivity

$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$
$f(n) = \Theta(f(n))$

symmetry

$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$

transitivity

$f(n) = X(g(n))$ & $g(n) = X(h(n))$
$\downarrow$
$f(n) = X(h(n))$

complementarity

$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

$f(n) = o(g(n)) \iff g(n) = \omega(f(n))$

---

(5) Lemmas, formulae and techniques

upper bound trick commonly done also to relate terms

↳ substitute something that is bigger to get a more uniform expression

eg. $\lg n! = (\lg(n)(n-1)\dots(1)$
$= \lg(n) + \lg(n-1)\dots$
$\leq \lg n + \lg n \dots$
$= n\lg n$

eg. $n^2 + 3n\lg n$
$\leq n^2 + 3n^2$
$= 4n^2$
$\leq 4e^n$

arithmetic series

$$1+2+3\dots n = \frac{n(n+1)}{2}$$

$$a_1 + a_2 \dots a_n = \frac{n(a_1+a_n)}{2}$$

geometric series

$$\sum_{i=0}^{n} a\cdot r^i = a\left(\frac{1-r^{n+1}}{1-r}\right)$$

harmonic series

$$\sum_{k=1}^{n}\frac{1}{k} = \frac{1}{1}+\frac{1}{2}\dots\frac{1}{n} = \ln n$$

sum of squares

$$\sum_{k=1}^{n} k^2 = 1+4+9\dots n^2 = \frac{n(n+1)(2n+1)}{6}$$

sum of cubes

$$\sum_{k=1}^{n} k^3 = 1^3 + 2^3 + 3^3 \dots n^3 = \frac{n^2(n+1)^2}{4}$$

logarithmic rules ✡ super useful!

$$\log_a b = \frac{\lg b}{\lg a}$$

$$x^k = a \longleftrightarrow k = \log_x a$$

$$\lg(ab) = \lg a + \lg b$$

$$\lg \frac{a}{b} = \lg a - \lg b$$

$$\lg a^b = b \lg a$$

integral summation trick

↳ when sum of functions, increase bound by 1, use definite integrals.

eg. $\sum_{i=a}^{n} f(i) = f(a) + f(a+1) \dots$

$$\leq \sum_{i=a}^{n+1} f(i)$$

$$\leq \int_a^{n+1} f(i)\, di$$

factorial

$$\lg \lg n + \lg \lg (n/2) + \lg \lg (n/4) \dots$$

$$= \lg \lg n + \lg(\lg n - 1) + \lg(\lg n - 2) \dots$$

$$= \lg(\lg(n) \times (\lg n - 1) \times \lg(n-2) \dots) \Rightarrow \lg(\lg n)!$$

---

⑥ correctness of iterative algorithms

↳ in simple and deterministic algorithms, proving correctness is often trivial.

↳ loops are a common construct in most algorithms — how can we prove their correctness?

loop invariants for a loop running for N iterations, a loop invariant is a sequence of assertions $S_1, S_2 \dots S_N$ such that each S_i holds at the start of the ith iteration and such that S_N implies the desired correctness property.

1. initialisation: it is true prior to the first iteration of the loop
2. maintenance: if it is true before the start of the ith iteration, it remains true before the start of the $i^{th}+1$ iteration
3. termination: when the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct

0. define some property
1. base case: loop initializes it to _____. show that invariant is true.
2. induction step: assuming invariant is true for i^{th} loop, show that with values carried over, invariance holds for $i+1^{th}$ loop
3. termination: algorithm terminates when _____ (cases?), show that invariant is still correct after termination. summarise result.

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

Claim 3.1.1 (Inner loop invariant). *Let A' be the array A before line 5 in an outer loop iteration with $j = J$. At the start of an inner loop iteration with $i = I$, the following hold:*

1. $A[1..I] = A'[1..I]$.
2. $A[I+2..J] = A'[I+1..J-1]$.
3. *All elements in $A[I+2..J]$ are greater than $A'[J] = key$.*

Proof. The proof is by induction. The claim is trivial for the base case $I = J-1$. Suppose the claim holds for I, and the inner loop continues to $I-1$, we want to argue for $I-1$. By the induction hypothesis, at the start of the run with $i = I$,

(1) $A[1..I] = A'[1..I]$

(2) $A[I+2..J] = A'[I+1..J-1]$

(3) $A[I+2..J] > A'[J] = key$

Suppose I continues to $I-1$. Condition 1 holds true because: $A[1..I] = A'[1..I]$ and (1) gives $A[1..I-1] = A'[1..I-1]$. Condition 2 holds true because: $A[I] = A'[I]$ (from condition 1) and $A[I+1] = A[I]$ (after an inner loop iteration), therefore $A[I+1] = A'[I]$, combined with (2) gives $A[I+1..J] = A'[I..J-1]$, which is equivalent to $A[(I-1)+2..J] = A'[(I-1)+1..J-1]$. Condition 3 holds true because: $A'[I] = A[I] > key$ and $A[I+1] = A'[I]$ (after an inner loop iteration), therefore $A[I+1] > key$, combined with (3) gives $A[I+1..J] > key$, which is equivalent to $A[(I-1)+2..J] > key$. □

Suppose the inner loop terminates when $i = I$. This can happen because of two reasons, either $I = 0$ or $A[I] \le key$. In the former case, Conditions 2 and 3 of the inner loop invariant gives $A[2..J] = A'[1..J-1]$ and $A[2..J] > key$. In the latter case, the inner loop invariant gives: $A[1..I] = A'[1..I]$, $A[I+2..J] = A'[I+1..J-1]$ and $A[I+2..J] > key$.

Claim 3.1.2 (Outer loop invariant). *At the start of an outer loop iteration with $j = J$, $A[1..J-1]$ is the sorted list of elements originally in $A[1..J-1]$.*

Proof. The proof is by induction. The claim is trivial for the base case $J = 2$. Suppose the claim holds for J, and we want to argue for $J+1$. By the induction hypothesis, at the start of the run with $j = J$, $A[1..J-1]$ is a sorted arrangement of the original elements in these positions. Call this array A'. After the inner loop terminates in this run of the outer loop, by the paragraph after Claim 3.1.1, there can be two cases:

- $I = 0$. In this case, $key < A[2] \le \cdots \le A[J]$ where $A[2..J]$ is a right-shift of $A'[1..J-1]$.
- $A[I] \le key$. In this case, $A[1] \le \cdots \le A[I] \le key$ because these elements are the same as in A' and they were sorted, while $key < A[I+2] \le \cdots \le A[J]$ because these elements are a right-shift of $A'[I+1..J-1]$ which were sorted.

In either case, line 8 ($A[I+1] = key$) ensures that $A[1..J]$ is sorted and consists of the elements in $A'[1..J-1]$ and key. □

Applying Claim 3.1.2 for $J = A.length + 1$ proves the correctness of INSERTION-SORT.

Notes

1. comparing functions, tricks. $n^{f(n)} = 2^{\lg n^{f(n)}} = 2^{f(n)\lg n}$ $\Rightarrow$ compare powers!

2. $\lg\lceil(\lg n)!\rceil$ $\rightarrow$ translates to addtl. $\lg n$ $\Rightarrow$ $\lg n \cdot \lg\lg n$ because of stirling's approx.

Analysis of recursive algorithms

① Divide and conquer

↳ divide and conquer is an algorithm design paradigm that involves the following:

divide the problem into sub-problems that are smaller instances of the original problem

conquer the subproblems by solving them recursively (at some base level)

combine the solutions in a way that is correct to get a solution to the original problem

---

② correctness of recursive algorithms

↳ idea is based on induction. A recursive algorithm solves problems by splitting them into smaller problems until some base case is met, then combines them upwards.

↳ some recursive algorithms do the work at divide (eg quickSort, local maxima) while others do it at combine (eg. mergeSort)

↳ to prove correctness, in general, if applicable, using **strong induction** to show that:

1. **divide** step does indeed return the base case eventually, and maintains some invariant property (eg. quicksort — subarrays on left all ≤ pivot)
2. **conquer** step maintains some invariant property of base problems, based on divide that got it to reach base step (eg. mergesort, single element trivially sorted)
3. **combine** step (if applicable) maintains some invariant property, based on property of previous combine (eg. if subarrays sorted, merge will return another sorted array)
   - 3.1 show that combine is correct for base case (s)
   - 3.2 show that combine is correct for inductive steps by strong MI

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

```
MERGE(A, p, q, r)
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[8]

② conquer

① divide

invariant

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

③ combine

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k-p=0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p = r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

③ running time of recursive algorithms

substitution method

0. guess that $T(n) = X(n)$
1. assume divides one time. show that $T(n) = X(n)$ (induction step) for values of c and n_0

eg. $T(n) = T(n/2) + T(\lceil n \rceil) + n \Rightarrow$ assume $T(n/2) = X(n/2)$, $T(\lceil n \rceil) = X(\lceil n \rceil) \Rightarrow$ prove $T(n) = X(n)$

2. show that boundary solution — based on n_0 as base step for values of c and n_0

---

recursion tree

1. draw recursion tree, find no. of operations per layer

$(k)^h \cdot n = 1 \iff h = \log_k n$

(k): split eg. 2/3; 1: leaf

- imbalanced tree
- balanced tree

2. using height as bounds, argue for asymptotic bounds with

2.1 max height: longest simple path from root to leaf

2.2 min height: shortest simple height

2. compute summations as $h \cdot g(n)$

→ substution method for rigour

---

master method

$T(n) = a\,T(n/b) + f(n)$, $a \geq 1$, $b \geq 1$,

$n/b = \lfloor n/b \rfloor$ or $\lceil n/b \rceil$

1. Find $\log_b a = \frac{\ln a}{\ln b} = k$
2. cases

f(n) grows faster than height

$f(n) \geq c \cdot n^{k+\varepsilon} = \Omega(n^{k+\varepsilon})$, $\varepsilon > 0$

$a\,f(n/b) \leq c\,f(n)$ for some $c < 1$ and all sufficiently large n (regularity condition, ensures sum of subproblems < f(n), root dominates)

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

f(n) grows slower than height, height dominates

$f(n) \leq c \cdot n^{k-\varepsilon} = O(n^{k-\varepsilon})$, $\varepsilon > 0$

$\Rightarrow T(n) = \theta(n^k)$

f(n) grows about as fast as height, both height and f(n) matter

$c_1 n^k \leq f(n) \leq c_2 n^k = \theta(n^k)$

$\Rightarrow T(n) = \theta(n^k \cdot \lg n)$

useful tricks

$T(\text{base})$ undefined $\Rightarrow$ set $T(n_0) = q > 0$. set $c = \max/\min(q, c)$

Notes

1. given $T(f(n)) \Rightarrow$ manipulate

eg. $T(2^n) = T(2^0) + T(2^1) \dots T(2^{n-1}) + n^2$

$\Rightarrow T(2^n) - T(2^{n-1}) = T(2^{n-1}) + n^2 - (n-1)$; manipulate to get tractable expression

$\Rightarrow$ set $N = 2^n$

Then $T(N) = 2T(2^{n-1}) + 2n - 1$

$= 2T(N/2) + 2\lg N - 1$

2. height of T_n is $\lg\lg n$.

$\sqrt{\sqrt{T_n}} \dots = n^{\frac{1}{2^i}} = 2$

$\frac{1}{2^i} = \log_n 2$

$= \frac{\lg 2}{\lg n} = \frac{1}{\lg n}$

$2^i = \lg n$

$i = \lg\lg n$

3. change base to get familiar terms in log.

eg. $\frac{n^3}{\lg^2 n} + \frac{n^3}{(\lg n - \lg 3)^2} + \frac{n^3}{(\lg n - \lg 9)^2} \dots \Rightarrow$ change to base 3

4. factorize out

$T(n) = \dots$ (some n term) (some sum or product of n)

Show it converges to constant

thm $T(n) = \Theta(n \text{ term})$

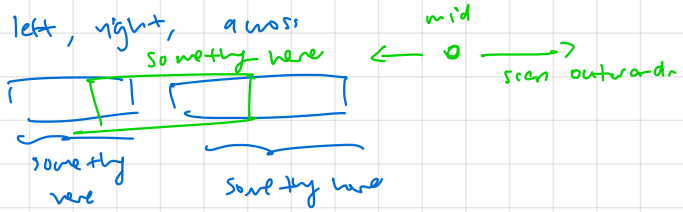
5. merge sort / merge subroutine is a good way to think about divide & conquer algorithms

divide → until ↓

conquer → trivially

combine → correctly by invariant / induction

6. useful divide & conquer paradigm : left, right, across



Some recursive algorithms

1D peak finding

2D peak finding

Probabilistic analysis

① tools of analysis

indicator variables

↳ provide a convenient way of converting between probability and expectations

$$X_i = \begin{cases} 1 & \text{if event } A \text{ occurs for some } i \\ 0 & \text{otherwise.} \end{cases}$$

$P(X_i = 1)$ = some probability of A_i

$$E(X_i) = P(X_i = 1)(1) + P(X_i = 0)(0)$$

$$E(X) = E\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} E(X_i)$$

linearity of expectations

$$E(X+Y) = E(X) + E(Y)$$

$$E(Y) = \sum_x E(Y \mid X = x) \cdot P(X = x)$$

$$E(aX) = a\,E(X)$$

geometric distribution

↳ $X \sim G(p_{success})$

↳ number of times to get 1 success

$$P(X = k) = \underbrace{(1-p)^{k-1}}_{k-1 \text{ failure}} \underbrace{p}_{1 \text{ success}}$$

$$E(X) = \sum_{k=1}^{\infty} (1-p)^{k-1} p \cdot (k)$$

reshape

$$= p\left[\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} \cdots\right]$$

(geometric series)

$$= p\left(\frac{1}{p} + \frac{(1-p)}{p} + \frac{(1-p)^2}{p}\right) \cdots$$

$$= 1/p \;//$$

$$X \sim G(p_{success}) \Rightarrow E(X) = 1/p$$

useful integrals

$$\int x \ln x \, dx = \frac{x^2 \ln x}{2} - \frac{x^2}{4}$$

② common analysis problems

counting occurrences

The hiring problem

suppose at all times, we want to best person for the job. We sequentially interview them, and hire if he is the best so far. With n interviews, how many will we hire?

1. define indicator variable

$$X_i = \begin{cases} 1 & i\text{th candidate is best so far} \\ 0 & \text{otherwise.} \end{cases}$$

probability changes w/ i

$P(X_i = 1) = \frac{1}{i}$, assuming random & i.i.d.

2. expectation over sequence

$$\sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$$

counting paired occurrences — the birthday problem

Q: in a room of n people, how many have the same birthday?

1. Define indicator variable

$$X_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ have same birthday}, i \neq j \\ 0 & \text{otherwise} \end{cases}$$

$$P(X_{ij} = 1) = \sum_{d=1}^{365} \left(\frac{1}{365^2}\right) = \frac{1}{365}$$

($\frac{1}{365}$: chance on any day for either person)

$$E(X_{ij}) = \frac{1}{365}$$

2. sum to count for room of n people

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

$$E(X) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(X_{ij})$$

$$= \left[(n-1) + (n-2) \cdots 1\right] \frac{1}{365}$$

$$= \frac{n(n-1)}{2} \frac{1}{365}$$

---

combining geometric variables — the coupon collector problem

There are n coupons. How many coupons to collect until we get k unique if each draw is random?

Idea: sum of different geometric RV.

1. i^{th} geometric RV

Let X_i be no. of times to get i^{th} unseen ball. Then $P_{success} = \frac{n-i+1}{n}$ ← unseen / ← total

$$X_i \sim b\left(p = \frac{n-i+1}{n}\right)$$

2. how long to get all k?

$$\sum_{i=1}^{k} E(X_i) = \sum_{i=1}^{k} 1 \Big/ \frac{n-i+1}{n}$$

$$= \sum_{i=1}^{k} \frac{n}{n-i+1} \Longleftrightarrow \frac{n}{i}$$

symmetry!

$n-1+1, n-2+1 \cdots n-n+1$

$1, 2 \cdots n$

$$= n \sum_{i=1}^{k} \frac{1}{i} \leq n \lg k$$

Randomised algorithms

① randomised algorithms : motivation

1) probabilistic analysis

↳ best and worst case analysis give us bounds on our algorithm's performance

↳ but with knowledge of the input distribution, we can analysis the average performance of the algorithm, given the probability & performance of the algorithm

⇒ problem : how do we know the distribution of inputs ?

2) worst case inputs and adversaries

↳ suppose the worst case time is very bad, and we have an adversary. eg. quicksort given sorted sequence.

⇒ can we take control away from adversaries ?

3) inject randomness into algorithms

↳ by enforcing randomised bottlenecks at points in our algorithm, we enforce an a priori distribution and take control away from adversaries.

⇒ we sacrifice certainty for controlled uncertainty

4) types of randomised algorithms

Las vegas algorithms — These algorithms are always correct. Their running time is a random variable that is bounded in expectation.
eg. randomised quicksort

Monte Carlo algorithms — correctness is a random event — they may give an incorrect output with some small probability.

---

② average case analysis of Las vegas algorithms

↳ idea : we inject controlled randomness into the input. we then analyse the expected running time over these inputs.

correctness

1. show that randomisation preserves validity of input
2. prove correctness as usual

running time

let x be input drawn from some distribution D_n. $x \sim D_n$.

$A(n) = \underset{x \sim D_n}{E}$ (running time of A on x)

1. consider types of input and their probability of occurrence.
2. For each type of occurrence, analyse running time
3. formulate into expectation $A(n) = \sum_{x \sim D_n} A(x)\, p(x)$

average case analysis of quick sort

0. suppose we have an input of length n of distinct elements. Then for any input, we observe that the execution of quick sort depends on the rank of a given element, in ascending order.

① divide into types

1. Let S_i be the set of permutations of input such that x_i, the ith ranked element is the start (and is thus picked as the pivot)

1.1 observe that $P(S_i) = \sum_{i=1}^{n} \frac{1}{n!} = \frac{1}{n}$

② analyse individual case

2. let $G(i)$ be the average running time of quick sort when x_i is the pivot.

$$G(i) = \underbrace{(n-1)}_{\text{comparisons to partition}} + \underbrace{A(i-1) + A(n-i)}_{\text{recursive calls to either half, excluding pivot}}$$

③ formulate weighted sum. solve if necessary

3 $A(n) = \sum_{i=1}^{n} P(S_i) \cdot G(i)$

$= \frac{1}{n} \sum_{i=1}^{n} G(i)$ since uniform distribution, $P(S_i) = \frac{1}{n}$

$= \frac{1}{n} \sum_{i=1}^{n} \underbrace{(n-1)}_{\text{comparisons to partition}} + \underbrace{A(i-1) + A(n-i)}_{\text{recursive calls to either half, excluding pivot}}$

$= (n-1) + \frac{1}{n} \sum_{i=1}^{n} A(i-1) + A(n-i)$

observe that $= \underbrace{A(0) + A(1) \dots A(n-1)}_{\text{equal}} + A(n-1) + A(n-2) \dots A(0)$

$$= n-1 + \frac{2}{n}\sum_{i=1}^{n} A(i-1)$$

$$= n-1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$

useful trick to prep for substitution

3.1 We guess that $A(n) = O(n \lg n)$

$$A(n) = n-1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$

$$\leq n-1 + \frac{2(c)}{n}\sum_{i=1}^{n-1} i \lg i$$

$$\leq n-1 + \frac{2c}{n}\int_{i=1}^{n} i \lg i \, di$$

(n) increase by 1 now integrate

$$= n-1 + \frac{2c}{n}\left[\frac{i^2 \ln i}{2} - \frac{i^2}{4}\right]_1^n$$

$$= n-1 + \frac{2c}{n}\left[\frac{n^2 \ln n}{2} - \frac{n^2}{4} + \frac{1}{4}\right]$$

$$= n-1 + cn\ln n - \frac{cn}{2} + \frac{c}{2n} \leq c \cdot n \ln n \quad \forall c \geq 2.$$

Hashing

① hashing

1) hash functions

↳ the idea behind hashing is to drastically reduce the size of the direct access table.

$h : U \longrightarrow \{0, 1, \ldots m-1\}$

2) collisions and chaining

↳ suppose $h(x) = h(y)$, $x \neq y$. we call this a (collision)

↳ (chaining) we handle collisions by adding an element to a linked list of an item maps to the same value as another item

3) complexity of hashing and chaining

↳ for a hash table of size m and N elements, hash table is of size $O(m+N)$. m pointers to linked list, total of N elements.

4) properties of good hash functions

1. minimize collisions. query(x) and delete(x) need to traverse the LL, and so take $\theta(|h(x)|)$
2. minimise storage space, $M = o(N)$, N is stored items.
3. $h(x)$ should be easy to compute

---

② universal hashing

1) motivation

↳ if U is large such that $|U| \geq M(N-1)$, then $\forall\, h : U \longrightarrow [m]$, there exists a set of N items to be stored that have the same hash value.

⇒ for a large enough n, an adversary can always force the worst case scenario by picking N items to store that map to same hash value

(proof)
1. pigeonhole principle. suppose not. That there exists a h that maps all $|U|$ items to the hashtable s.t. every slot at most $N-1$ items.
2. then total $M(N-1) < M(N-1)+1$ items. contradiction.

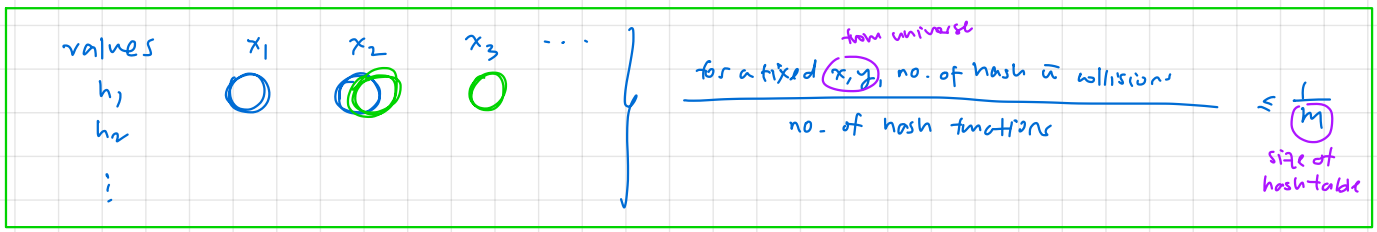
2) universal hashing

↳ key idea : randomization. Fool the adversary by not fixing the hash function!

universal hashing Suppose $\mathcal{H}$ is a set of hash functions $h_i : U \to [m]$. Then we say $\mathcal{H}$ is universal if $\forall x \neq y$,

$$\frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

$\Rightarrow$ for any pair of x, y where $x \neq y$, if h is chosen uniform randomly from $\mathcal{H}$, then there is at most $1/m$ probability that $h(x) = h(y)$



$\Leftrightarrow$ for a given pair x, y s.t. $x \neq y$, the proportion of $h \in \mathcal{H}$ that would make $h(x) = h(y)$ is at most $1/m$

3) properties of universal hashing

1. For $x_1, x_2 \ldots x_n \in U$, E(number where x_N (fixed) collides w other elements) $< \frac{N}{m}$

$\Leftrightarrow$ if I have inserted $N-1$ elements, the expected no. of collisions when I insert the next element is bounded by $\frac{N}{m}$, since there are at most N distinct x_i.

for $i < N$, let C_i $\begin{cases} 1 & \text{if } x_N \text{ collides w } x_i \\ 0 & \text{otherwise} \end{cases}$

$E(C_i) \leq \frac{1}{m}$. $\sum_{i=1}^{N-1} E(C_i) \leq \frac{N-1}{m} < \frac{N}{m}$ //

2. $m > N \longrightarrow E$(cost of N operations) $= O(N)$

E(cost of 1 operation) $= O(1) + E$(no. of collisions when hashing x_i)

if $M > N$, then $M > N > i$, where i is the i^{th} insertion. Then for the i^{th} operation, E(collisions) < 1, and E(insertion of i^{th} element) $= O(1)$

E(cost of N operations) $= \sum_{i=1}^{N} E$(cost of 1 operation) $= N \times O(1) = O(N)$

3. $M > N \longrightarrow E(\#\text{ of collisions for } x_i \in N) < 2N$

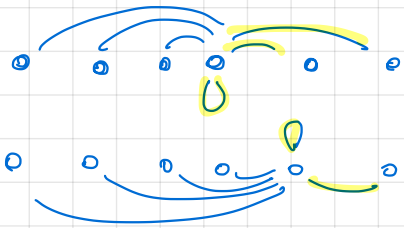
$\Leftrightarrow$ if $M > N$, then although at the i^{th} insertion we expect less than 1 collisions, there could still be some collisions over a <u>sequence</u> of N insertions. So the <u>total</u> number of collisions across N insertions is bounded by $2N$.

At the i^{th} insertion, we have inserted $i-1$ elements, and that could cause a collision. We could find the total number by finding the bound at the i^{th} insertion and then summing them to N.

An easier alternative is to recognise that in N elements, there are a fixed number of paired permutations that could be drawn that would cover every collision scenario, and use that as a bound instead.

true bound: only backwards

permutation bound: both directions and itself



Let $A_{ij} \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases}$

$$E\left(\sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}\right) = \sum_{i=1}^{N} E(A_{ii}) + \sum_{i \neq j}^{N} E(A_{ij}) \leq \underbrace{N}_{\text{will collide}} + \underbrace{N(N-1)}_{{}_NP_2 = \frac{N!}{(N-2)!} = N(N-1)} \cdot \underbrace{\frac{1}{m}}_{\text{1/m chance of collision}} < 2N$$

since $M > N$

4. $M > N \longrightarrow$ expected maximum no. of elements hashed to a given slot $< \lceil 2N \rceil$

$\Leftrightarrow$ if $x_1, x_2 \ldots x_N$ are hashed, and M/N, we expect that a given slot will have at most $\lceil 2N \rceil$ elements hashed into it

In the worst average case, all the <u>expected</u> collisions will be hashed to the same slot — that is our expected maximum load case.

if M/N, we <u>expect</u> that the <u>total</u> no. of collisions $< 2N$. If all to the same slot, and recognising that any <u>pair</u> of elements at the same slot contributes to a collision, the expected no. of elements at the slot is $\sqrt{2N}$.

intuition of not being 2 for loops to pair all @ same slot

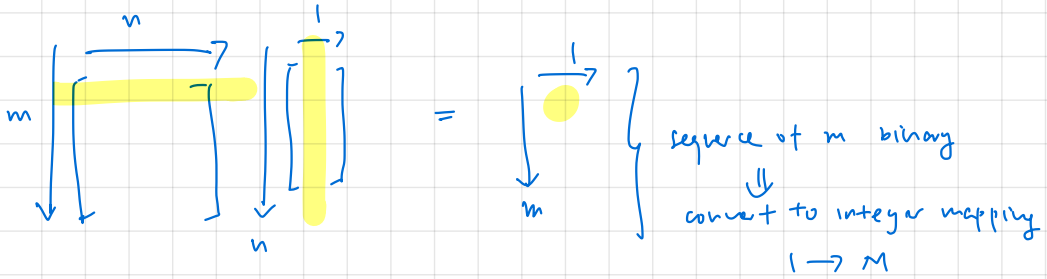
$$E(\max) = E\left(\sqrt{\max^2}\right) \leq \sqrt{E(\max^2)} \leq \sqrt{E(\text{total. collisions})} \leq \sqrt{2N}$$

③ constructing universal hash functions

1. let $u = |U|$ and m be powers of 2, where $|U| = 2^u$, $M = 2^m$.
   Then we can represent any $x \in U$ as a binary sequence of length u.

2. Then for a binary matrix $A \in \{0,1\}^{m \times u}$, we define h as

$$h_A(x) = A x \text{ (element-wise mod 2)}$$



3. Then $\mathcal{H} = \{ h_A : A \in \{0,1\}^{m \times u} \}$, where A is any binary $m \times u$ matrix.
   There are $2^{m \times u}$ possible h_A.

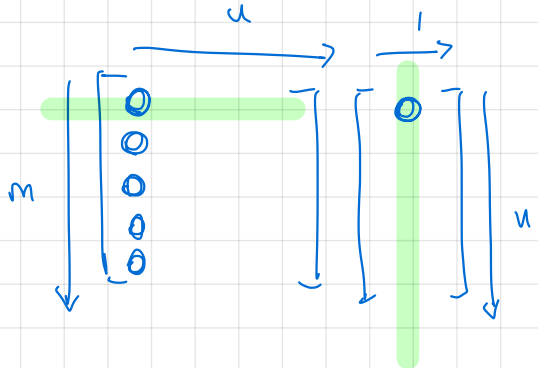
proof that $\mathcal{H}$ is universal

0. collision occurs when $A(x) = A(y) \iff A(x-y) = 0$, though $x \neq y$

1. For convenience, let $z = x - y$. so we seek to show $P(Az = 0) \leq \frac{1}{m}$

   1.1 $x, y \in \{0,1\}^u$. i.e. x and y are distinct binary sequences of length u.
   obviously, $z \neq \vec{0}$.

   1.2 observe that to produce $\vec{0}$, a given entry i of the u-bit input multiplied by a column in A must perfectly offset that of the sum (mod 2) of all other columns. (assuming the entry is non-zero)



$$A^i = -\sum_{j \neq i} z_j A^j \bmod 2$$

A^i: column vector; $\sum_{j \neq i} z_j A^j$: sum of all other column vectors

2. Since A is uniformly chosen in $2^{m \times u}$ possibilities, the choice of the (i,j) position is independent of all other positions $\longrightarrow$ and the i^{th} column also independent of all other columns

   2.1 so the probability that A^i is chosen to exactly offset sum of others, given a fixed z, where $z = x - y$ & $x \neq y$, is $\frac{1}{2^m} = \frac{1}{m}$ exactly
   (entire column)

   2.2 So $\mathcal{H}$ is universal

complexity of universal hashing

↳ the space required to specify a particular hash function is $O(um) = O(\log|u| \log|m|)$.

↳ time complexity is that of matrix multiplication

---

④ pairwise independent hashing

pairwise independent family If is a pairwise independent family of hash functions $h : U \to [m]$ if it has the property that

$$P_{h \sim H}(h(x) = i_1, h(y) = i_2, x \neq y) \leq \frac{1}{m^2}.$$

$\Rightarrow$ the probability that a given (x, y) pair map to values $i_1, i_2 \leq \frac{1}{m^2}$

for each distinct $x, y \in U$
  for each i_1, i_2 they map to
    $\frac{\text{no. of times } (i_1, i_2) \text{ appear}}{\text{no. of hash functions}} \leq \frac{1}{m^2}$

→ also no. of mappings, if you think about it

pairwise → universal

$$P_{h \sim H}(h(x) = h(y)) = \sum_{i=1}^{m} P(h(x) = h(y) = i) \leq m \cdot \frac{1}{m^2} \leq \frac{1}{m} \Rightarrow \text{universal}$$

m possible values to be equal

hash $x_1, x_2 \ldots x_N$, E(no. of elements to j) $< \frac{N}{m}$ ← after $1 \ldots N$, expected @ given slot

1. $C_i = \begin{cases} 1 & \text{if } h(x_i) = j \\ 0 & \text{otherwise} \end{cases}$ if hash to that position ← i.e. collisions pertaining specifically to j

From pairwise independence, we know $P(h(x) = i_1, h(y) = i_2) \leq \frac{1}{m^2}$.

$h(y) = i_2$: we don't care, so we sum over them (law of total probability)

$$P(C_i = 1) = \sum_{i=1}^{m} P(h(x) = j, h(y) = i)$$

$$\leq m \cdot \frac{1}{m^2} = \frac{1}{m}$$

2. N elements. E(hashes to j) $= \sum_{i=1}^{N} E(C_i) \leq \frac{N}{m}$ ← for each insertion, whether will go to j

⑤ perfect hashing

↳ idea: suppose we know a priori the set of $x_1 \dots x_N$ elements we want to hash. Can we construct a hash function so that query(x) can be answered in O(1) worst case time?

1) do perfect hash functions exist?

↳ observation: if we pick an appropriate m, then the E(total collisions for $x_1 \dots x_N$) for family of universal hash functions < 1

↳ previously:

Let $A_{ij} = \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases}$

$$E\left(\sum_{i=1}^{N}\sum_{j}^{N} A_{ij}\right) = \sum_{i=1}^{N} E(A_{ii}) + \sum_{i \neq j}^{N} E(A_{ij}) \leq \underbrace{N}_{\text{will collide}} + \underbrace{N(N-1)}_{NP_2 = \frac{N!}{(N-2)!} = N(N-1)} \cdot \underbrace{\frac{1}{m}}_{\text{1/m chance of collision}}$$

$$\sum_{i \neq j}^{N} E(A_{ij}) \leq N(N-1) \cdot \frac{1}{m}$$

$E(\text{collisions}) < 1$ if $m = N^2$ ⇒ there must exist at least one $h \in \mathcal{H}$ that $E(\text{collisions}) = 0$, since expectation is less than 1.

ie. when $x \neq y$ but $h(x) = h(y)$

2) double hashing for O(N) space complexity

↳ we know that for any $x_1 \dots x_N$, there exists at least one perfect hash function, and we need $m = N^2$ to get that.

↳ But $O(N^2)$ space is large. If we set $m = N$, we are going to get collisions.
⇒ since m grows by a square, why not use perfect hash at second level?

1. choose $h : U \to [m = N]$ from $\mathcal{H}$.
2. Let L_k be the number of x_i for which $h(x_i) = k$, $k \in [m = N]$
   size of each collision chain
3. We know that for each bucket, there must exist a universal function $h_k : [L_k] \to [L_k^2]$ such that there are no collisions among the L_k elements mapped there by h initially
4. total space complexity = $\underbrace{O(N)}_{\text{original table}} + \underbrace{O(N)}_{\text{sum of all secondary tables}}$

   4.1 $A_{ij} = \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases}$ $\quad \sum_{k}^{N} L_k^2 = \sum_{i}^{N}\sum_{j}^{N} A_{ij}$ → pairs! same line of reasoning as $\lceil 2N \rceil$ vs. $2N$

   4.2 if $m > N$, $\sum_{k}^{N} E(L_k^2) \leq N + N(N-1) \cdot \frac{1}{N} < 2N$ — by choice of $m = N$

⑥ division hashing

prime uniformly selected from $1 \ldots K$

definition Given a prime number p, $h_p : \mathbb{Z} \to \{0, \ldots p-1\}$, $h_p = x \bmod p$

value → p slots

choice of p to avoid adversarial arguments, we choose p uniformly from $1 \ldots K$.

CHOOSE-PRIME(K)

1 Choose random number p uniformly from $\{1, \ldots, K\}$
2 **if** p is prime
3 **return** p
4 **else** CHOOSE-PRIME(K)

1. There are $> \frac{N}{\lg N}$ prime numbers from $1 \ldots N$
2. probability that prime is chosen in line 1 $> \frac{N/\lg N}{N} = \frac{1}{\lg N}$.

2.1 viewing as Bernoulli random variable, expected no. of trials before first success is $< \lg K$.

$\frac{b \lg K}{K}$ collision bound

Suppose we have $0 \le x, y < 2^b$ Then if we choose p uniform randomly from $1 \ldots K$

collision when $h_p(x) = h_p(y) \Rightarrow (y - x) \bmod p = 0$

So $z < 2^b$ must be divisible by one of its prime factors.

We know that for $z < 2^b$, there are $t < b$ prime factors.

since there are $> K/\lg K$ primes from $1 \ldots K$ and p is uniformly chosen,
$P(p$ is one of the $< b$ prime factors of $z) < b \cdot \frac{\lg K}{K}$

applying division hash to strings

↳ the division hash takes in integers.
↳ how do we convert a sequence of characters from alphabet $\{0, 1, \ldots l-1\}$ to an integer?

from $0 \ldots l-1$

power of base

$$x = \langle x_1, x_2, x_3, \ldots x_m \rangle \quad int(x) = \sum_{i=1}^{m} x_i \cdot \underbrace{l^{m-i}}_{\text{base}}$$

conversion to base-l bit representation

$int(x) < 2^b$, $b = m \lg l$

Notes

universal hashing & E(no. of pairs that collide)

$$X_{ij} = \begin{cases} 1 & h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases}$$

$P(X_{ij} = 1) = \frac{1}{m} = E(X_{ij})$, m is no. of slots, by universality

$$E(X) = {}^{n}C_2 \cdot E(X_{ij}) = {}^{n}C_2 \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m}$$

Pattern matching

① Monte Carlo equality checks with hashing

↳ recall that monte carlo algorithms are correct w a certain probability.

↳ also observe that uniform and division hashing can both have their probability of collisions controlled by some hyperparameter. In division hashing, controlled by sampling range of p $1 \dots k$.

i) equality checking

```
if h(x) = h(y)
    return true
return false
```

hashed word size

when we use division hashing, we map to $[p] \le k$ since p is chosen from $1 \dots k$.

Then we need $\lg k$ bits to represent the hash value. often $\lg k = O(\lg N)$, by choice of k.

ii) error rate

0 false negative rate $\quad x \neq y \Rightarrow h(x) \neq h(y)$. obvious.

False positive rate

1. let x and y be length-m sequences from alphabet $\{0, 1 \dots \ell - 1\}$.
2. Then $x = int(X)$, $y = int(Y) < 2^b$, $\boxed{b = m \lg \ell}$
3. Then we know $P(h_p(x) = h_p(y) \mid x \neq y) < b \cdot \frac{\lg k}{k}$
4. If we want $\boxed{b \cdot \frac{\lg k}{k} \le \varepsilon}$, we set $\boxed{k = \frac{2m}{\varepsilon} \cdot \lg \ell \cdot \lg\left(\frac{2m}{\varepsilon} \lg \ell\right)}$

(m: length of sequence; ε: error rate; ℓ: alphabet size)

② Rolling hashes w division hash

↳ suppose we want to repeatedly hash adjacent sequences. Then repeated hashings of $T(s+1 \dots s+m)$ to $T(s+2 \dots s+m+1)$ would take $O(m)$ time every time, even though they only differ by 1 character.

linearity of int. conversion

$$int(T[s+1 \dots s+m]) = \sum_{i=1}^{m} T(s+i)\, \ell^{m-i}$$

$$\Rightarrow int(T[s+2 \dots s+m+1]) = \ell \cdot int(T[s+1 \dots s+m]) - \ell^m \cdot T[s+1] + T[s+m+1]$$

(everything ↑ by power 1; remove first char, ℓ^m; add new char, who is ℓ^0)

linearity in modulo operation

$(a+b) \bmod p = [(a \bmod p) + (b \bmod p)] \bmod p$

$c \cdot a \bmod p = c \cdot (a \bmod p) \bmod p$

$$h_p(T[s+1 \dots s+m+1]) = h_p\left(\ell \cdot h_p(T[s+1 \dots s+m]) - h_p(\ell^m) \cdot T[s+1] + T[s+m+1]\right)$$

(next: $s+1$; previous: s)

---

③ pattern recognition

1) naive pattern recognition

↳ 1D array of length n, looking for sequence of length m.

```
for i in 1 to n-m+1          // O(n)
    isEqual = T[i, ... i+n-m+1] == patter // O(m)
```

$O(nm)$

2) karp-Rubin algorithm

(idea) we use hashing to make equality check run in $O(1)$ time, and rolling hashes to hash next in $O(1)$ time. Then the algorithm runs in $O(n)$ time.

```
KarpRubin(T, P, f, l)
```

(T: sequence, P: patter, f: error, l: alphabet)

n = T.length

m = P.length

$f' = f/n$ // by union bound

$K = \frac{2m}{f'} \cdot \lg l \cdot \lg\left(\frac{2m}{f'} \lg l\right)$

p = choose-prime(K)

$h_p(x) = x \bmod p$ // set hash function

hash-pattern = h_p(int(P)) // hash patter

hash-text = h_p(int(T[1 ... m])) // hash first m

```
for s = 1 to n-m
    if hash-patter == hash-text
        return
    else
        hash text = hp(l·hash-text - T[s+1]·hp(l^m) + T[s+m+1]) // update
```

FPR for a single comparison is bounded by f.

$E_i = \begin{cases} 1 & \text{if false positive} \\ 0 & \text{otherwise} \end{cases}$

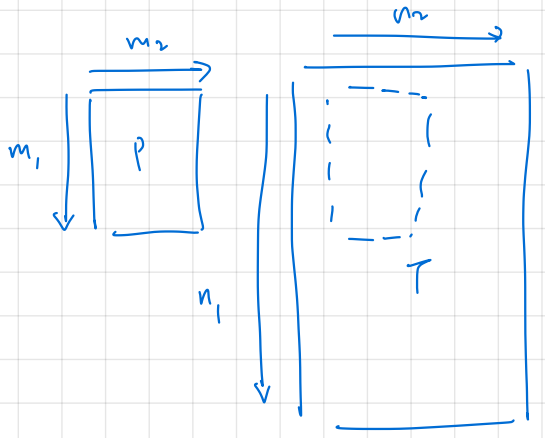
$P(E_i = 1) < f$

By union-bound, since probability at most as large as sum (where all mutually exclusive)

$$\sum_{s=0}^{n-m} P(E_s = 1) < \frac{f}{n} = f'$$

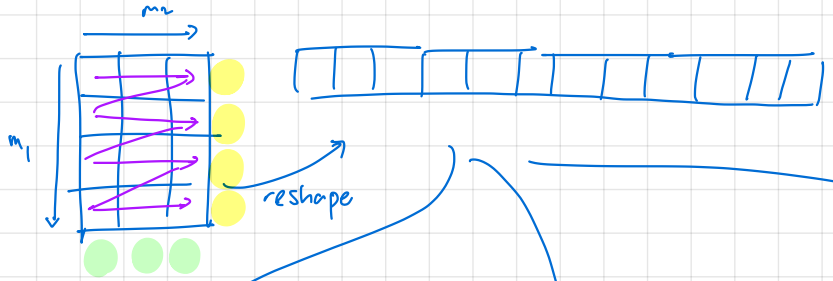
3) Karp Rubin algorithm in 2D

idea: we build on the idea in the 1D case by reshaping the blocks into a 1D sequence, and accordingly adjusting our rolling hash



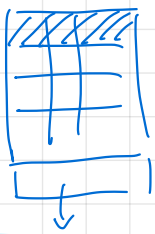
↳ in the 1D case, we convert sequences into base ℓ by:

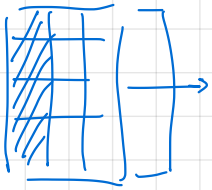
↳ in the 2D case, we can convert blocks by thinking of them as sequences



how to hash sequence? | how to update row-wise? | how to update column wise?

$$\text{int(region)} = \sum_{i=0}^{m_1-1} \sum_{j=1}^{m_2} x_{ij} \cdot \ell^{m_1 \times m_2 - (i \times m_2 + j)}$$





FP rate: we will run total of $o(n_1 n_2)$ times. $\delta_1 = \frac{\delta}{n_1 n_2}$

runtime: $o(m_1 m_2)$ + $o((n_1 - m_1 + 1) m_2)$ + $o((n_2 - m_2 + 1) m_1)$

$o(m_1 m_2)$: initial hashes; $o((n_1 - m_1 + 1) m_2)$: row wise updates; $o((n_2 - m_2 + 1) m_1)$: column wise updates

+ $o((n_1 - m_1 + 1)(n_2 - m_2 + 1))$: equality checks

Streaming

① streaming model

stream a sequence of insertions and deletions of items from a universe U. we will consider U to be the set $\{1 \dots |U|\}$.

stream can be : Add(3), Add(1), delete(3) ...

→ sequence

② frequency estimation

frequency estimation — f_i : number of times i occurs at end of stream.
no. insertions of i − deletions of i

⇒ can we design a smart way to occupy small amt. of space but query efficiently f_i?

- **hashing-based frequency estimation**
- direct access table. $O(|U|)$.
- BST $O(m)$, where m is no. of distinct

idea : use hashcodes, maintain counter at each hashcode. increment & decrement when add or delete

INITIALIZE(U, k)
1 Choose hash function $h : [U] \to [k]$ from a universal family $\mathcal{H}$
2 Create table T of k entries, each initialized to be 0

ADD(i)
1 $T[h(i)] = T[h(i)] + 1$

DELETE(i)
1 $T[h(i)] = T[h(i)] - 1$

QUERY(i)
1 **return** $T[h(i)]$

Then we know that will only over estimate in some error.

$$f_i \le E(\hat{f}_i) \le f_i + m/k$$

(m: no. of distinct; k: hash table size)

1. by turnstile assumption, for every element, no. insertion $\ge$ no. deletion
   - 1.1 so $E(\hat{f}_i) \ge f_i$
2. $X_{ij} = \begin{cases} 1 & \text{if } h(i) = h(j) \\ 0 & \text{otherwise.} \end{cases}$
   - 2.1 $\hat{f}_i = \sum_{j=1}^{u} f_j X_{ij}$ — so i either collides in all other or itself
3. if $i = j$, $X_{ij} = 1$, if j, $P(X_{ij} = 1) \le 1/k$ by universality of h
   - 3.1 so $E(\hat{f}_i) = E\left(\sum_{j}^{u} f_j X_{ij}\right) = f_i + f_j \sum_{j \ne i} E(X_{ij}) \le f_i + \sum_{j \ne i} f_j \cdot \frac{1}{k} \le f_i + \frac{m}{k}$

(bounded by $\le m$ in equation)

4. If we set $k = 1/\varepsilon$, $\varepsilon > 0$:

$$f_i \le E(\hat{f}_i) \le f_i + m/k \iff f_i \le E(\hat{f}_i) \le f_i + m\varepsilon.$$

space complexity

hash table

since there are m distinct, need $\lg m$ bits to index each

$k = \frac{1}{\varepsilon}$ rows, each row with $\lg m$ bits

hash function

A is a $|U| \times |M|$ matrix, so $\lg|U| \cdot \lg|m|$ size.

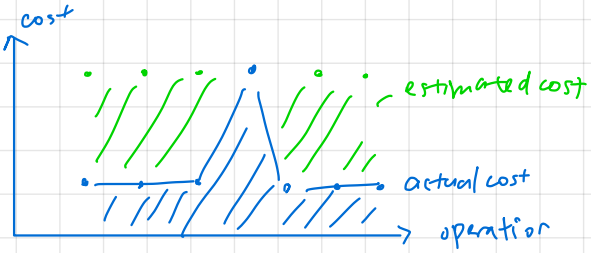
$O\left(\left(\frac{1}{\varepsilon} + \lg|U|\right)\lg m\right)$ bits $\approx O\left(\frac{1}{\varepsilon}\right)$

Amortized analysis

① amortized analysis

↳ in analysing a sequence of operations on a data structure, an easy way to compute the total worst-case cost is to take the individual worst case costs and multiply

↳ but you may be overcounting!



we use amortised analysis to get tighter deterministic bounds on a sequence of operations

amortised analysis a strategy for analysing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence may be expensive

⇒ does not involve probability; guarantees the average performance of each operation (in a known sequence) in the worst case

often worst case sequence

---

② aggregate method

technique we show that for all n, a sequence of n operations takes worst case time $T(n)$. Then the amortized cost is $\frac{T(n)}{n}$.

---

③ accounting method

idea impose an extra charge on cheap operations and "save it", use that guaranteed savings (show that never dips below zero) to pay for later expensive operations, during which you charge 0

(which is almost certain)

1. formulate $c(i) = \begin{cases} \dots \\ \dots \end{cases}$ for diff. types of operations → overestimate cheap, build credit; → underestimate ex., consume credit
2. show that $\sum_i c(i) \geq \sum_i t(i)$, that is, total amortized cost bounds total actual cost → directly, or that "savings" ≥ 0 always
3. $c(i)$ gives amortised cost of diff. operation types

useful visualization using table

operation | actual cost | amortized cost | effect on bank

can represent one in terms of other — eg. insert(n) => insert(1), n times

excess => save

withdrawal of actual cost

show that cheap operation's effects on bank cover for expensive operation's effect on bank

technique figure out how much you need to overpay per unit up front so that later operation (that can happen, since this is happening) is paid for.
Technique gives avg. worst case cost of each operation type

---

(4) the potential method

idea use a potential function modelling the state of the data structure to track excess / withdrawals of "credit", rather than using explicit operation amortised costs to track "credit"

1. $c(i) = t(i) + \phi(i) - \phi(i-1) = t(i) + \Delta\phi(i)$
   amortised of i^{th} operation is actual cost + change in potential after operation
   ie. some adjustment to the true cost tracking excess / shortfalls from previous operations implicitly

2. show that by design of representative ϕ, $\phi(n) - \phi(0) \geq 0$

3. then total cost bounded by $\sum_1 c(i) = \sum t(i) + \phi(n) - \phi(0) \geq \sum t(i)$.

4. then amortised cost of each operation is respective $c(i)$

why does the technique work?

1. we set amortised cost $c(i) = t(i) + \phi(i) - \phi(i-1)$

   1.1 then sequence of n operations, total amortised:

$$\sum c(i) = \sum_{i=1}^{n} t(i) + \phi(i) - \phi(i-1)$$

$$= t(n) + \phi(n) - \phi(n-1)$$
$$+ t(n-1) + \phi(n-1) - \phi(n-2)$$
$$\vdots$$
$$+ t(2) + \phi(2) - \phi(1)$$
$$+ t(1) + \phi(1) - \phi(0)$$

$$= \sum_{i=1}^{n} t(i) + \phi(n) - \phi(0)$$

1.v then so long as $\phi(n) - \phi(0) \geq 0$, $\sum_{i=1}^{n} t(i)$ bounded by $\sum_{i=1}^{n} c(i)$

2. then we can find the amortized cost of <u>each</u> of the i operations

Technique Design a potential function such that using <u>state of data structure</u>

1. ϕ increases in small increments $\bar{w}$ cheap operations
2. falls by large amounts $\bar{w}$ expensive operations

↳ $t(i) + \Delta\phi(i)$ small

→ spot traits to design ϕ

3. $\phi(0) = 0$ (for convenience)

$\Rightarrow$ then $c(i) = t(i) + \Delta\phi(i)$ gives cost of <u>specific</u> operation

---

⑤ application to dynamic tables

i) motivation for dynamic tables

↳ we do not always know how many objects will be stored

↳ when not enough space, we create new larger table, copy into it

↳ when too much space, we create new smaller, copy into it

$\Rightarrow$ how can we design choices of new table size and <u>when</u> to do it, so that we have reasonable amortized time complexity?

2) dynamic expansion

idea everytime table is full, double the size

amortised cost of n insertions

1. set $\phi(i) = 2i - \text{size}(T)$. let us analyse each i of the n insertions.
2. case 1: table is full when trying to insert i

2.1 then $c(i) = \underbrace{i}_{t(i)} + (2i - \underbrace{2(i-1)}_{\text{new table}}) - (2(i-1) - (i-1))$, copy i-1, put 1

$= i + 2 - 2i + 2 - i + 1$

$= 3$

3. case 2: table is not full when trying to insert i

3.1 then $c(i) = 1 + (2i - 2T) - (2(i-1) - 2T)$

$= 3$

4. in both cases, $c(i) = 3 = O(1)$

OP accounting method : $c(i) = \begin{cases} (i-1)+1 & \text{if full and doubling} \\ 1 & \text{otherwise.} \end{cases}$

then $\sum_{i=1}^{n} c(i) = n + \sum_{j=0}^{\lg n - 1} 2^j$ ← doubling

$\leq 3n$

3) dynamic expansion and contraction

idea table full, double; @ 1/4 capacity, halve

piecewise potential function!

1. set $\phi(i) = \begin{cases} 2 \cdot T.num - T.size & \text{if load} \geq 1/2 \\ T.size/2 - T.num & \text{if load} < 2 \end{cases}$

2. analysis of table insert
3. analysis of table delete

} see CLRS pg. 470

Notes

1. idea: cost of examine & operate

need to iterate!

instead, we maintain a pointer ⇒ then we can go to pointer & operate, reset pointer if necessary ⇒ costs 2 each time

Dynamic programming

① recursion and optimal substructure

optimal substructure we say that a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems

⇒ i.e. fancy way of saying that you can write a recursive expression for the solution that is correct

notion of optimality the choice of "optimal" is a misnomer, in the sense that it refers more to correctness, just often in the context of an optimization problem

⇒ eg. Fib(n) = Fib(n-1) + F(n-2) exhibits optimal substructure, since the "optimal" solution is the correct solution.

proving optimal substructure "cut sub-optimal subproblem solution, paste optimal"

1. formulate a recurrence relation of a problem eg. T(n) = 1 + T(n-1)
   - 1.1 solution to the problem consists of making a choice (eg. where to split), leaving one or more subproblems (divide) to be solved, then combined (conquer).
2. (suppose not) suppose T(n) is an optimal solution, and suppose T(n-1) is not.
   - 2.1 (show contradiction) then we could improve T(n) by T(n) = 1 + T*(n-1)
   - 2.2 but T(n) is optimal, and cannot be improved. contradiction.

we formalize the notion of "recursion is correct" with "exhibits optimal substructure"

have optimal substructure ⟺ can be solved w divide and conquer

---

② Recursive problems with overlapping subproblems

↳ in recursive algorithms, we can represent the computation graph as a recursion tree. But in some recursive formulation, some nodes are identical — that is, there exists repeated computation

overlapping subproblems when a recursive algorithm revisits the same independent problem repeatedly, we say that the problem has overlapping subproblems.

↳ note that independent = do not share resources = can be executed in parallel = a unique node

③ Dynamic programming

tackling repetition in divide & conquer

↳ divide and conquer algorithms partition the problem into subproblems, solve them recursively, then combine their solutions. They exploit optimal substructure to get a solution better than brute force.

↳ when a divide & conquer algorithm is inefficient (not incorrect) is where there exist overlapping subproblems, because of repeated computation

⇒ that's where DP techniques come in

idea brute force, but carefully to avoid repeated computation

memoization

idea memorize the natural, but inefficient recursive algorithm.

↳ if value not in table, compute & add it. Else just use

running time $O($ n independent problems × complexity of solving/combining $)$

correctness by induction

bottom up implicit memoization

idea compute subproblems from the base case, and proceed in order of dependency

↳ essentially converting recursion tree to graph where each node is unique subproblem, solving in topological order

technique

1. find recurrence, prove optimal substructure
2. initialize base cases
3. for loops starting from base cases to n, each loop combining by using recurrence

running time running time of iterative algorithm

correctness

- induction
- loop invariant
  - ↳ value always computed before (indirectly)

④ summary

1- formulate recurrence
  1.1 break into subproblems
  1.2 think how to choose among subproblems (eg. max, min, if $T(n-1)$)

2. prove optimal substructure

3- point out overlapping subproblems
  3.1 (count unique, if possible)

4- bottom up
  4.1 initialize table w base cases
  4.2 loop, starting w case right after base
  4.3 terminate by return

5. prove correctness by
  5.1 loop invariant of optimal solution to subproblem always being computed
  5.2 combination gives optimal

knapsack

Given $(v_1, w_1), (v_2, w_2) \dots (v_n, w_n), W$.

Find a subset of $S = \{1 \dots n\}$ such that $\sum_{i \in S} w_i \leq W$

main idea of subset:
if can include → compare includes. not
if cannot → don't, skip
base case

1. 0. enumeration of 2^n subsets is costly.

1. we can formulate the recurrence in the regular knapsack problem.

$$T(n, w) = \begin{cases} 0 & \text{if } n = 0 \text{ or } w = 0 \\ \max\left(T(n-1, w)^{\text{skip}}, T(n-1, w-w_n)^{\text{include}} + v_n^{\text{value}}\right) & \text{if } w_n \leq w \\ T(n-1, w) & \text{otherwise no choice but to skip} \end{cases}$$

Greedy algorithms

① Greedy choice property & implications

greedy choice property A locally optimal choice is a globally optimal choice

1) proving that a problem exhibits it

idea: prove that greedy choice is always in an optimal solution

⟺ greedy choice never leaves you worse off

1. Formulate a recurrence that greedily chooses a subproblem to solve using some measure (eg. latest activity)
2. Let $S(n)$ be an optimal solution.
   - 2.1 suppose $S(k)$, where k is greedy choice is not in any optimal solution
   - 2.2 show that swapping $S(k)$ with choice leaves solution equal or better
   - 2.3 Then it must be in at least one optimal solution

we use idea to show that while we may not get a specific solution, our solution in this choice is always optimal

2) exploiting the greedy choice property

↳ observe that DP algorithms improve o.i. divide & conquer problems with overlapping subproblems by memoizing (explicitly or implicitly)

↳ but efficiency can be improved further if we reduce the no. of subproblems we need to solve → how do we choose them?

⇒ if a problem exhibits the greedy choice property, we can exploit it by always making the greedy choice and reduce the no. of subproblems to solve

---

② Greedy algorithms

↳ in a greedy algorithm, we choose the solve subproblem(s) that seem best at the moment, and solve only those that are chosen

↳ the choice(s) made may depend on choices so far, but it cannot depend on any future choices or solutions to subproblems

greedy vs. DP algorithms

↳ in DP, we solve subproblems, and then combine them. often by making some choice among the solutions.

↳ in greedy, we choose first which subproblems to solve, and then go ahead with it, pruning the computation graph

idea brute force, but even more carefully, by choosing wisely less subproblems

formulating a greedy algorithm

1. formulate recurrence, one for optimal, one for greedy
2. prove optimal substructure ⇒ can recurse
3. prove greedy choice property ⇒ can recurse this smart way
4. recursive algorithm + time complexity

optimal substructure optimal solution exactly uses optimal to sub this way ↳ ie. can reuse, overlapping

greedy choice there exists an optimal solution where direct choice is inside

1. Let S be an optimal solution (sequence).
2. Suppose greedy choice not inside
   - 2.1 always some x_j it can swap with.
   - 2.2 swapping always no worse off

Reductions

① computation problems

problems a computational problem is defined by a set of (x, s) tuples where x is a valid input instance) and s is a set of valid solution(s) to the input x

decision problems a problem where for any valid input x, the solution is either a YES or NO

search problems general problems where the solution goes beyond YES and NO

---

② input size & running time

encodings An encoding of a set S of abstract objects is a mapping e from S to a set of binary strings

↳ either encoding is explicitly specified, or we follow conventions

integer ⟶ binary
string ⟶ concatenation of constant-bit length strings
graph ⟶ adjacency matrix

running time we say that an algorithm solves a concrete problem (i.e. encoded explicitly) in $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm produces a solution in $O(T(n))$ time.

↳ take note that we size of encoding, not value, not abstract representation

---

③ reductions

Reductions between decision problems

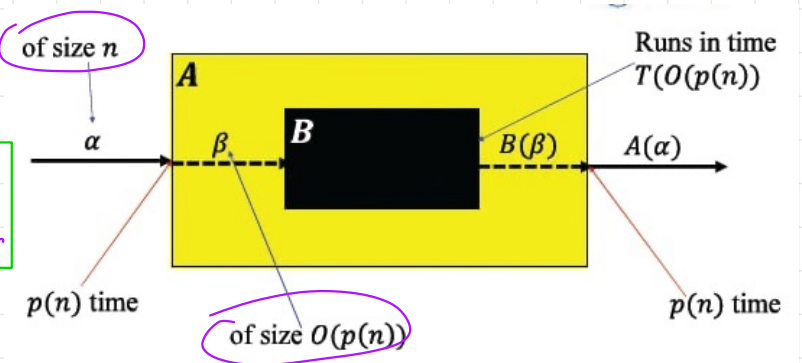
Given two decision problems X and Y, we say that X reduces to Y if there exists an algorithm that takes an input x of problem X and outputs an instance y of problem Y, such that x is a YES instance of X iff. y is a YES instance of Y.

reductions in polynomial time

if transformation algorithm A reduces x to y in polynomial time, we say X reduces to Y in polynomial time. $X \leq_p Y$

Running time composition

If there is a $p(n)$ time reduction from A to B, and a $T(n)$ time algorithm to solve problem B on instances of size n, then there is a $T(O(p(n))) + O(p(n))$ time algorithm to solve problem A on instances of size n.



$T(p(n)) + 2\,p(n)$
solving — to and from

polytime transform → b must be at most of size $O(f(n))$

proof that A reduces to B in $p(n)$ time

1. transformation algorithm of $a \rightarrow b$
   1.1 proof of run in $p(n)$ time. If obvious, state obvious.
2. YES a ⟶ YES b
3. YES b ⟶ YES a

---

④ decision reduces to optimization

↳ given an optimization problem, we can convert it into a decision problem

⇒ is there a solution w value ≤ k ? ⇒ get solution ⟶ compare to k

---

⑤ pseudo-polynomial time reduction

↳ an algorithm that runs in polynomial time in the numeric value of the input (but exponential in size of input encoding) is pseudo polynomial

eg. $O(n) \rightarrow O(2^{\lg n}) = O(2^{\text{encoding size}})$ for integers, w size $\lg n$

$O(n) \rightarrow O\left(\frac{n \lg n}{\lg n}\right) = O(\frac{1}{\lg n}$ encoding size$)$ for arrays, w size $n \cdot \lg n$

⑥ Reductions & relative hardness

1. $A \leq_p B$.
2. If there is a $p(n)$ time algorithm for B, there is one for A.
   2.1 proof by construction. Reduce, solve, transform.
3. If there is <u>no</u> $p(n)$ time algorithm for A, then there is none for B
   3.1 proof by contradiction. None for A, have for B. Then can convert to B and solve in $p(n)$ time. contradiction.

---

⑦ general reduction

Karp-Levin reduction Direct reduction to solve.

1. $X \xrightarrow{R_1} Y$, $Y \xrightarrow{R_2} X$, both run in $p(n)$ time.
2. then if A is algorithm to solve Y, $R_2(A(R_1(x)))$ is a solution to X.

Cook reduction A more general type of reduction where we invoke an algorithm A_Y for Y as a subroutine a polynomial <u>number of time</u> in order to solve X.

P and NP

① P and NP

1) definitions

P complexity class The set of problems that can be solved in polynomial time.

NP complexity class The set of problems for which polynomial time verifiable certificates of YES instances exist.

↓

efficient p(n) certifier a polynomial time algorithm that outputs YES iff. input is a solution to problem X.

↳ ie. correct solution → YES
wrong solution → NO ⟺ YES → correct solution

2) key results

P ⊆ NP proof by construction : just return the solution and see if same.

show X is in NP

1. propose algorithm that takes the solution (of size p(n)) and outputs YES / NO
   - 1.1 in p(n) time
   - 1.2 correctly : YES ⟺ solution is correct

---

② NP hard and NP complete

NP hard A problem is said to be NP hard if for every problem B in NP, B can be reduced to it. $\forall B \in NP,\ B \leq_p X$.

showing NP hard

1. propose a reduction of NP hard problem Y to X.
2. show it runs in p(n) time.
3. show YES x → YES y
4. show YES y → YES x
5. By transitivity of reductions, since Y is NP hard, X is NP hard.

NP complete A problem that is in NP (verifiable in $p(n)$) and also NP hard.

---

③ Some NP complete problems

independent set Given a graph G and integer k, is there a subset of k vertices such that no two are connected by an edge?
ie. max no. of vertices such that no two connected

vertex cover Given a graph G and integer k, is there a subset of k vertices such that each edge is incident to at least one vertex
ie. smallest no. of vertices such that all edges included

set cover Given integers k and n, and a set S of subsets of $\{1 \ldots n\}$, are there $\leq k$ of these subsets whose union equals $\{1 \ldots n\}$?
ie. minimal joing of subsets to get original set

3 SAT Given a CNF formula where each clause contains exactly 3, not necessarily distinct literals, does it have a satisfying truth assignment?

partition Given a set of positive integers S, can the set be partitioned into 2 sets of equal sums?

Hamiltonian path Given a graph G, is there a path in a directed graph from some beginning vertex to an ending vertex, such that every node is visited exactly once?

Clique A set of vertices U of a graph G is a clique if every pair in U has an edge in G. Given a graph G and integer k, is there a fully connected subgraph with $\geq k$ nodes?

---

④ reductions covered in class

independent set $\leq_p$ clique

1. Given graph G. Take the complement graph of G' as an input to the clique problem. set $k' = k$.
2. Obvious that this transformation runs in polynomial time, $O(V^2)$.
3. YES independent set → YES clique
   - 3.1 Let S be independent set solution.
   - 3.2 Then corr. k nodes will be disconnected in complement graph and form a clique of size $k' = k$.
4. YES clique → YES independent set
   - 4.1 symmetrical argument to 3.

---

independent set $\leq_p$ vertex cover

can modify G, or modify k!

1. Given graph G and k to IS. We check that vertex cover of size (n-k). graph G'.
2. obvious that this runs in polynomial time. $O(V^2)$
3. YES independent set → YES vertex cover
   - 3.1 Let solution to IS be S. Then G-S is a solution to vertex cover.
   - 3.2 proof: Let (u,v) be every edge in G. Then since IS, u,v cannot both be in S.
4. YES vertex cover → YES independent set
   - 4.1 Let solution n-k nodes to VC be S. ie. there is a solution ⇒ YES instance
   - 4.2 Then G-S is solution to IS problem. proof by contradiction: if edge (u,v) both u,v in G-S, then edge not covered in S. contradiction.

useful way to think about vertex cover

vertex cover $\leq_p$ set cover

defining a suitable transformation such that solution to one is exactly solution to other

1. For a given graph $G(V, E)$ and k to the VC problem, we generate the following as input to the SC problem
   - 1.1 $n = |E|$
   - 1.2 $S_k = \{$ edges incident on vertex $k \}$
   - 1.3 Then $\mathcal{F} = \{S_k\}_{k \in V}$
2. It is obvious that the generation runs in polynomial time. $O(V^2)$.
3. YES VC $\rightarrow$ YES SC
   - 3.1 Let S be the solution to VC, with k vertices.
   - 3.2 Then we know all edges are covered in S. By definition of S_k, the union of S_k, $k \in S$ will be $\mathcal{F}$.
4. YES SC $\rightarrow$ YES VC
   - 4.1 By definition, the subsets that form solution S have corr. nodes, which cover all edges in G.

3SAT $\leq_p$ independent set

1. input to 3SAT as $x_1 \dots x_n$ in k clauses of 3 literals each. Then we generate an input to IS with each x_i as a vertex, and connect an edge between all literals in the same clause, all literals to negations of itself
2. obvious that this runs in polynomial time.
3. YES IS $\rightarrow$ YES 3SAT
   - 3.1 Let s be independent set of size K. idea: use edge to constrain simultaneous presence
   - 3.2 Then S cannot contain vertices labelled by conflicting literals, since they are connected by an edge.
   - 3.3 S must contain exactly one vertex from each clause, since exactly one can be chosen from a given triangle (so that no connected vertices) idea: use edge to isolate one from each clause
   - 3.4 Thus it is possible to choose a set of literals that are present in every clause and have no contradictions, and setting these literals to true will make the 3-CNF statement true

4. YES 3SAT ⟶ YES IS

4.1 For every triangle, pick exactly one vertex that is true in the assignment satisfying the 3SAT.

4.2 k clauses. Valid assignment means at least 1 true in each clause, with no contradictions

4.3 Then 4.1 will create an independent set, since x_i and $\sim x_i$ cannot simultaneously be in solution (ie. no edge for $x_i - \sim x_i$) and choosing one per triangle also means no edges.

---

⑤ Notes about reductions

1. 3SAT → boolean expressions that are logically equivalent
   → as a graph

2. IS → edges as constraints (cannot both be present in solution)
   → $(u, v) \in E$, u & v cannot both be in solution, since connected

3. VC → $(u, v) \in E$, at least one of u, v must be in solution

4. Set cover → by definition, S_k = {some property}, union{S_k} = {original}

5. Clique → complement to IS

decide

6. partition / subset sum → pad w $x_i = 0$
   → double the set w repeat

7. longest simple path / cycles → pad w dummy vertices that are longer than what could originally exist

3 SAT $\leq_p$ hamiltonian cycle

3 SAT $\leq_p$ subset sum