Lecture 2: Asymptotic Analysis

Definition of algorithm:

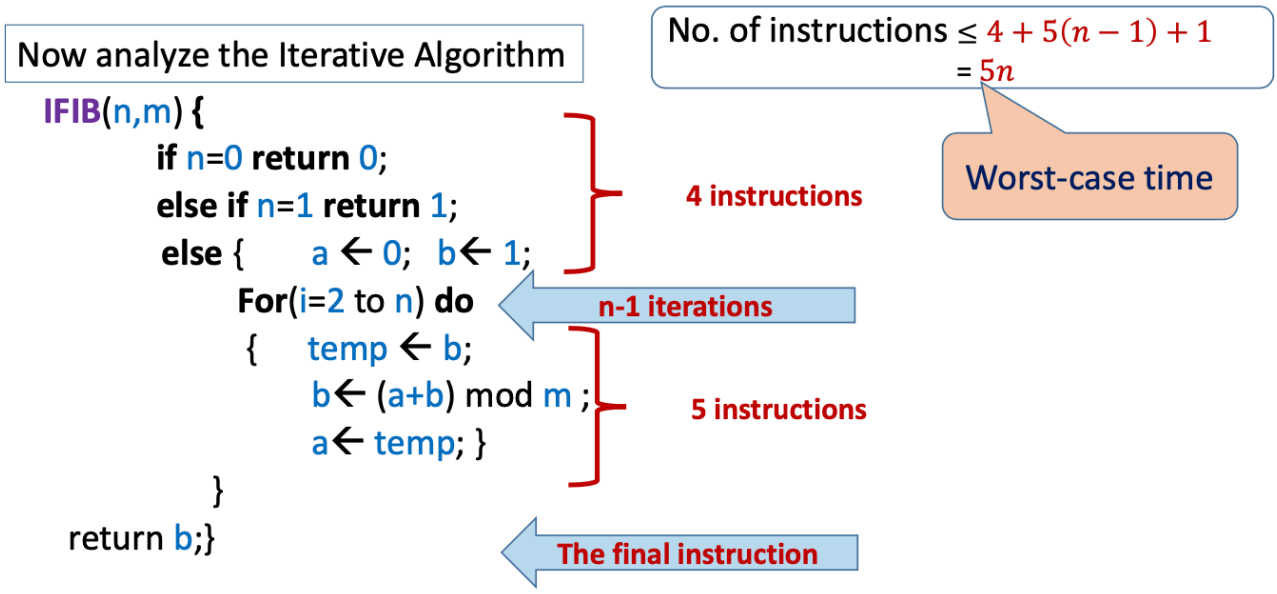
A well defined finite instruction to solve a given computational problem.

Word-RAM model

A model of computation that a random-access machine can do bitwise operation on a single word of w bits.

- It works with size w bits, which means it can store integer up to 2^w
- Word size matches problem size(for a problem of size n, $w \geq \log n$). The model allows bitwise operations such as arithmetic and logical shifts to be done in constant time.
- The number of possible values is *U*, where$\boldsymbol{U \leq 2^w}$. (This is important, this means the result of a programs never exceed the word size)

Count number of operations(Example):



Asymptotic Analysis

Estimate the rate-of-growth of runing time of an algorithm:

- O-notation (Upper bound)
- Ω-notation (Lower bound)
- Θ-notation (Tight bound)

O-notation

We defininе $f(n) = O(g(n))$ If $\exists c, n_0 > 0 : 0 \leq f(n) \leq c \cdot g(n)$ for $n \geq n_0$

Ω-notation

We define $f(n) = \Omega(g(n))$ if $\exists c, n_0 > 0 : f(n) \geq c \cdot g(n) \geq 0$ for $n \geq n_0$

IMPORTANT: O-notation is an upper-bound notation. It makes no sense to say f(n) is at least $O(n^2)$.

Θ-notation

We define $f(n) = \Theta(g(n))$ if $\exists c_1, c_2, n_0 > 0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, for all $n \geq n_0$

o-notation

We define $f(n) = o(g(n))$ if $\forall c > 0$: $\exists n_0 > 0 : f(n) < c \cdot g(n)$ for all $n \geq n_0$

ω-notation

We define $f(n) = \omega(g(n))$ if $\forall c > 0 : \exists n_0 > 0 : f(n) > c \cdot g(n)$ for all $n \geq n_0$

Limit theorem

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty : f(n) = O(g(n))$ (Intuitively, if $f(n) = \omega(g(n)) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, takes contrapotively)
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty : f(n) = \omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 : f(n) = \Omega(g(n))$ (Intuitively, if $f(n) = o(g(n)) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, takes contrapotively)
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 : f(n) = o(g(n))$
- $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty : f(n) = \Theta(g(n))$

Relationship between $o, O, \omega, \Omega, \Theta$

- $f(n) = O(g(n)) \to f(n) \neq \omega(g(n))$
- $f(n) = \omega(g(n)) \to f(n) \neq O(g(n))$
- $f(n) = \omega(g(n)) \to f(n) \neq \Theta(g(n))$
- $f(n) = \Omega(g(n)) \to f(n) \neq o(g(n))$
- $f(n) = o(g(n)) \to f(n) \neq \Omega(g(n))$
- $f(n) = o(g(n)) \to f(n) \neq \Theta(g(n))$
- $f(n) = \Theta(g(n)) \to f(n) \neq o(g(n))$
- $f(n) = \Theta(g(n)) \to f(n) \neq \omega(g(n))$

Properties of Big-O

- Transitivity:
  - $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \to f(n) = \Theta(h(n))$
  - $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \to f(n) = O(h(n))$
  - $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \to f(n) = \Omega(h(n))$
  - $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \to f(n) = o(h(n))$
  - $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \to f(n) = \omega(h(n))$
- Refletive:
  - $f(n) = \Theta(f(n))$
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$
- Symmetric
  - $f(n) = \Theta(g(n)) \to g(n) = \Theta(f(n))$
- Complementary

- $f(n) = O(g(n)) \rightarrow g(n) = \Omega(f(n))$
- $f(n) = o(g(n)) \rightarrow g(n) = \omega(f(n))$

Some concrete example:

1. Adversial wordle
2. Fibonacci number:
    - Recursive algortihm
    - Iterative algorithm
3. Sorting
    - Insertion sort

Lecture 3: Iterative algorithm

Iteractive algorithm: Algorithm with one or multiple loops, sequentially operating on the user input.

Proof of correctness of iteractive algortihm

- True before the first iteration
- If it's true before one iteration, it remains true after the iteration
- If it's true after the last iteration, it implies the correctness of the algorithm

Invariant: a condition is true at the begining of the iteration and is correctly maintained through the whole algorithm and can imply the correctness at the end of algorithm.

Some concrete example:

1. **Sorting**
    - Insertion sort: prove the correctness of insertion sort

```
INSERTION-SORT(A[1..n])
1.    for j = 2 to n
2.        key = A[j]
3.        // Insert A[j] into sorted seq 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
```

Proof:

1. **Step 1: what is the variant?**

   Here the variant is "A[1...j-1] is the sorted list of elements of A[1...j-1]"

2. **Step 2: Prove that the invariant is true before the first iteration**

   A[1] is trivially sorted before the first iteration

3. **Step 3: Prove that if the invariant is true before one iteration, it holds true after the iteration**

   3.1 Assume A[1...j-2] is sorted of elements originally in A[1...j-2],

   3.2 Line 6 ensures that any elements larger than A[j-1] are shifted exactly one place right

   3.3 Line 8 assigns value of A[j-1] to the position got by shift

   3.4 It can be implied from 3.2 that every element on the left of the position is smaller than the new value and every element on the right of the position is larger than the new value.

   3.5 Any violation of position to the sorted position will violate at least one of 3.1, 3.2, 3.2, 3.3, and 3.4

   3.6 Therefore the array is sorted after the iteration

4. **Step4: Prove that when the algorithm terminates, the invariant proves the corretness of algorithm**

   4.1 After the the first iteration the array A[1...n] is sorted by the invariant, which proves the correctness of the algorithm.

Analysis of the time complexity of interative algorithm

Usually simple, just find the operation in O(1) first and then check number of excution of O(1) operation in nested loops from center to outside.

Divide-and-Conquer(Recursively)

- Divide: divide a problem into subproblems that are the same problem but with less instance
- Conquer: If the problem is small enough to solve, solve it directly
- Combine: Combine the solution to subproblem to the solution of the original problem.

Proof of correctness of recursive algorithm

- Use strong induction
    1. Prove the correctness of the base case
    2. Assume the algorithm works for all smaller cases, prove that the algorithm holds true for a bigger case

Analysis of time complexity of recursive algorithm

- Analyze on the time complxity of conquering(the time complexity where the algorithm can be solved direcly), usually O(1)
- Analyze on the time complexity relations on each combination: that is:
    - The time complexity of a problem = Sum of time complexity of all subproblems + time complexity to combine the subproblems
- There are several common forms of relation ship:
    - $T(n) = a \cdot T(n-b) + f(n)$
    - $T(n) = a \cdot T(n/b) + f(n)$
    - $T(n) = a_1 \cdot T(n-b_1) + a_2 \cdot T(n-b_2) + f(n)$
    - $T(n) = a_1 \cdot T(n-b_1) + a_2 \cdot T(b_2) + f(n)$

Ways to solve a recurrence:

- **Recursion tree**: draw the tree and
- **Master method**: for format of $T(n) = a \cdot T(n/b) + f(n)$
- **Substitution method**: Guess and prove with math induction

Master's method

Compare $f(n)$ with $n^{\log_b a}$

- Case 1: if $f(n) = O(n^{\log_b(a)-\epsilon})$ (i.e. f (n) grows polynomially slower than $n^{\log_b a}$)

  then $T(n) = \Theta(n^{log_b a})$
- Case 2: if $f(n) = \Theta(n^{log_b(a)} \cdot \lg^k n)$ for some constant $k > 0$

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

- Case 3: if $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$

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

  Regularity condition(another condition of Case 3):

  if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant c<1

Substitution method

1. Guess the time complexity
2. Verify by induction

Concrete Example:

- ## HANOI-Tower

```
HANOI(n,src,dst,tmp):
  while n>0:
    HANOI(n-1, src, tmp, dst)
    MOVE the nth disk from src to dst
    HANOI(n-1, tmp, src, tmp)
```

Proof of correctness:

RTBD

Analysis of time complexity:

- $T(n) = 2 \cdot T(n-1) + 1$

  Claim: $T(n) = 2^n - 1$

  Proof: RTBD

- ## MERGE-SORT

```
MERGE-SORT(A,1,n):
  if (q>n):
    SORT A with insertion sort
  else:
    q = (1+n)/2
    MERGE-SORT(A,1,q)
    MERGE-SORT(A,q+1,n)
    MERGE(A,1,q,n)
```

Analysis of time complexity:

- $T(n) = 2 \cdot T(n/2) + \Theta(n)$

> 📔 **NOTE:**
>
> In terms asymptotic analysis, it doesn't matter for $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$, can just use (n/2)

- ## Powering a number

Problem: $f(n, m) = a^n \mod m$

- Instance: $O(\log n), O(\log m)$

Observation: $f(a + b, m) = f(a, m) * f(b, m)$

$O(n)$ **solution (divide and conquer):**

1. Divide: $f(n, m) = f(n - 1, m) * f(1, m)$
2. Conquer: f(1, m) can be solved in $O(1)$
3. Combine: $f(n, m) = f(n - 1, m) * f(1, m)$

$T(n) = T(n - 1) + O(1)$, which is $O(n)$

$O(\log n)$ **solution**

1. Divide:
   a. $f(n, m) = f(\lfloor n/2 \rfloor, m)^2$ if n is even
   b. $f(n, m) = f(1, m) \cdot f(\lfloor n/2 \rfloor)^2$ if n is odd
2. Conquer: $f(1, m)$ can be solved in $O(1)$
3. Combine:
   a. $f(n, m) = f(\lfloor n/2 \rfloor, m)^2$ if n is even
   b. $f(n, m) = f(1, m) \cdot f(\lfloor n/2 \rfloor)^2$ if n is odd

$T(n) = 2 \cdot T(n/2) + O(1)$, which is $O(\log n)$

Lecture 7: Amortised Analysis

Amortised analysis

Amortised analysis is the analysis of a sequence of operations to show that the average cost of each operation is small while some single operation may be expensive.

Definition of O(g(m,n)), or multiple-variable instance asymptotic analysis:

Given an function $f(m,n) \geq 0$, we say $f(m,n) \in O(g(m,n))$ if
$\exists c, n_0, m_0 > 0 \text{ such that } f(m,n) \leq c \cdot g(m,n)$ for all $n \geq n_0$ or $m \geq m_0$

Types of amortised analysis:

- Aggregate method: simple but less precise
- Accounting method: allowing specific amortised cost to each operation
- Potential method: allowing specific amortised cost to each operation

Accounting method(Banker's method)

Key idea:

- the amortised cost $c(i)$ is fixed for each operation while the true cost $t(i)$ Varies depending on when the operation is called;
- Pay extra amount for cheap operation as credit prepared in advance for the rare, expensive opeation.

Necessary condition to apply accounting method: $\sum_{i=0}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$, intuitively this means that the amount of money in the bank never drops below 0.

The total amortized cost provides an upper bound on the total true cost.

Potential method

- $\phi : \{S\} \rightarrow \mathbb{Z}_{\geq 0}$, where S is any state of computer or data structure
- $\phi(0)$: The initial state
- $\phi(i)$: The state after ith operation
- c : the amortised cost
- t : the true cost

$c(i) = t(i) + C \cdot (\phi(i) - \phi(i-1))$, where C is a constant across the analysis(The C may be omitted because the constant does not affect the time complexity of big O)

Necessary condition:

- $\forall i \in \{1, \ldots, n\} : \phi(i) \geq \phi(0)$

> 📔 **note:**
>
> Intuitively, the potential method means the depts that a state is accounted for but not paid yet. It's like counting the energy stored in that state

Concerete example:

1. Binary counter:

```
BINARY-COUNTER(A):
  i <- 0
  while i< A.length and A[i]==1:
    A[i] = 0
    i <- i+1
  if i < length[A]:
    A[i] = 1
```

The general objective of the analysis is to count the number of time of bit flipping.

- T(n) = total number of bit flipped during n increment
- Aim: get a tight bound for T(n):

Aggregate method

Define the state:

1. t(i): the number of flipings in ith increment
2. f(j): the number of flippings of jth bit
    - f(0) = n : every increment will flips bit 0
    - f(1) = n/2 : every two increments will flip bit 1 once
    - f(2) = n/4 : every four increments will flip bit 2 once
    - ...

$T(n) = \sum_{i=0}^{m} n/(2^i) = n \cdot \frac{1-\frac{1}{2^m}}{1-\frac{1}{2}} < 2n$

Hence the average cost of a single operation is O(1)

Accounting method

Define the amortised cost of each flips as follows:

- Flips from 0 to 1: 2
- Flips from 1 to 0: 0

The observation is that: each bit 1 in the the array has extra 1 in bank and this 1 can be used to set 1 to 0 later.

Claim: each increment only takes 2

proof: let i be the first 1 from right to left after the increment, then all the bits on right of i must be 1 before the increment, then all these bit 1s have extra 1 in the bank and hence can pay by themselves. Only the bit i should take 2 to increment from 0 to 1

Hence the amortized cost of n Operation$\leq 2 \cdot n = O(n)$

> 📔 **Conclusion 1:**
>
> Try find different ways to represent the states, can be the state of of operation executed, or certain part in the data structure.

> 📔 **Conclusion 2:**
>
> Try finding the frequency relationship between operations(usually try find whether #A<#B or B is rare but expensive and the cost of B is less than sum of previous A)

> 📔 **note:**
>
> When the cost and frequency of some special expensive operations are easy to count, it's easy to use aggregate method. For accounting method, if there is dependency of number of operations(especially when the expensive operation is proved to be less frequent than the cheap operaion, then can pay extra coin for cheap operation for expensive operation)

2. Queues

A queue structure that supporting the following two operations:

- INSERT: insert one element in O(1)
- EMPTY: empty the queue

Analysis of time complexity: the empty is a sequence of DELETE operation, where the number of delete equals to the number of elements in the queue. Moreover. it's assumed that DELETE is O(1) operation.

Aggregation method

Observation: The number of DELETE is less than the number of INSERT at any time

From the Observation we can claim that If there are k INSERT in the operation sequence, the sum of cost of all EMPTY will be $\leq$ k

Hence the total cost is $O(2k) = O(k)$, and the average cost of each operation is O(1)

#####

Accounting method

Define the amortized cost of operation as follows:

- INSERT: 2
- EMPTY: 0

  (Whenever insert, we pay extra 1, which is used to pay the empty operation later)

TOTAL_COST = 2 * #INSERT = $\leq 2n$

Potential method

Define the potential function as follows:

- $\phi(0) = 0$
- $\phi(i)$: the number of elements in the queue after i operations

Because the number of elements in the queue is never negative, hence $\phi(i) \geq \phi(0)$

- INSERT: $c_i = t_i + \phi(i) - \phi(i-1) = 1 + i - (i-1) = 2$
- EMPTY: $c_i = t_i + \phi(i) - \phi(i-1) = \phi(i-1) + 0 - \phi(i-1) = 0$

3. Dynamic table(vector in C++)

The size of array may not be known in advance, when the initial array is full, we should allocate more memory space. One way is to double the size of the array whenever it's ful.

Notation:

- n: number of element in the current state of table
- createTable(k): a system call that create a table of size k and return its pointer
- size(T): the size of table T
- copy(T,T'): copy the contents of Table T into T'
- free(T): free the space occupied by table T

Further explaination: RTBD(for review purpose):

1. Try discuss the time complexity of the data structure with aggregate method, accountign method and potential method respectively.

4. The ways to find the account:

Try to figure out the relationship that a group of element/operation should pay for the other group of element.

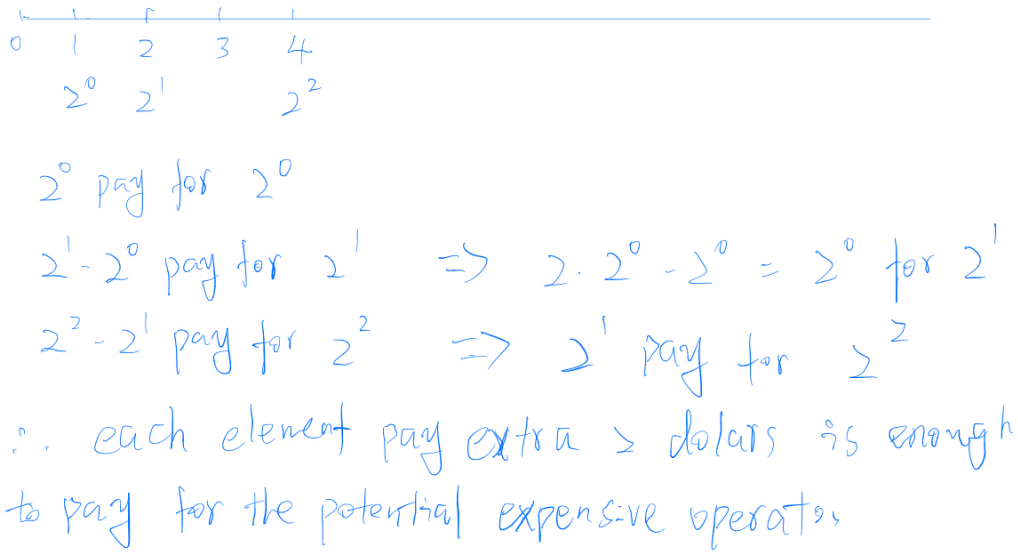
E.g.

17.2-2

Redo Exercise 17.1-3 using an accounting method of analysis.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.



The other example is when the cost is O(n) for kth operation where k is a Fibonacci number

Lecture 8: Dynamic Programming

Memoization

Storing the result of expensive function and return the result directly when called with the same input again.

Dynamic programming paradigm:

- Polynomial number of subproblems
- Huge overlap among the aubproblmes, which makes the recursive algorithm exponetial time
- Use memoization memorizing the result of sub-problems solved already

Steps to do dynamic programming

1. Characterize the structure of optimal solution
2. Recursively define the optimal solution

3. Compute the result of some base case(Bottom up manner)
4. Recursively compute the optimal solution given the optimal solution to subproblem.

Cut and paste

It's a kind of proof by contradiction used to prove optimal substructure. The generally goal is to show that optimal solution to constitute subproblem is necessary condition for an optimal solution to a problem. The format is as follows:

1. Assume exists an optimal solution to problem with suboptimal solution to the constitute subproblem
2. We can get a better solution by replacing the suboptimal solution to the subproblem with optimal solution to the subproblem, which contradicts the statement that the solution is optimal.

In proof, can just say "by cut and paste argument, a better solution can be obtained"

Concrete example:

Finbonacci Problem

- Recursive algorithm

```
mapping (int=>int) T
RFIB(n,m) {
  if (n==0) {
    return 0;
  } else if (n==1) {
    return 1;
  }
  if (!T[n-2]){
      T[n-2] = RFIB(n-2,m);
  }
  if (!T[n-1]) {
      T[n-1] = RFIB(n-1,m);
  }
  return T[n-1]+T[n-2];
}
```

- Iterative algorithm

```
IFIB(n,m) {
  mapping (int=>int) T;
  T[0] = 0;
  T[1] = 1;
  for (int i=2; i<n; i++) {
    T[i] = T[i-1] + T[i-2]
  }
}
```

The above program can be optimized to O(1) space complexity.

Longest Common Sequence

Difference between subsequence and substring:

- Subsequence: C is said to be a subsequence of A if we can remove zero or more elements from A to get C
- Substring: C is said to be a substring of A if $\exists i, j$ s.t. $C = A[i : j]$

Number of subsequence of a string:

Each bit-vector of length n determins a distinct subsequence of the string of length n, hence there are totally 2^n posssile distince subsequences

Checking whether a string C is a subsequence of B of size m takes O(m) time complexity. Hence a brute-force recursive algorithm takes $O(m \cdot 2^n)$ time in total

Check Dynamic Programming

Knapsack problem

Description:

Given instance $(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and a non-negative integer W, compute a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} v_i$ gets maximum under the condition that $\sum_{i \in S} w_i \leq W$

Dynamic programming solution:

- m[i,j] = 0, if i =0 or j = 0
- $m[i, j] = max\{m[i - 1, j - w_i] + v_i, m[i - 1, j]\}$, if $w_i \leq j$
- $m[i - 1, j]$, otherwise

Coin change

Description:

Given n cents, we need to change it with denominations $d_1, d_2, \ldots, d_k$. Goal is to use the fewest total number of coins.

- f(i): the minimum number of coin needed to change the amount i
- f(i) = ∞ initially for all $i \in \mathbb{Z}$
- f(i) = 0, if i = 0
- f(i) = $min_{j \in \{1,\ldots,k\}}[f(i - d_j)] + 1$

Rod Cutting

Given a rod with length of n and a vector indicating the price of rod with differentce length $P = (p_1, p_2, \ldots, p_n)$ with p_i indicates the price of rod with length i.

1. Characterize the structure of the optimal solution

   1.1 The structure of the optimal solution would be serveral cuts $r_1, r_2, \ldots, r_k$ with $\sum r_i = n$

   1.2 Given that $(r_1, r_2, \ldots, r_k)$ is the optimal cutting of n, $(r_1, \ldots, r_i)$ is the optimal cuttineg of length $\sum_{t=1}^{i} r_i$, otherwise can use cut and paste argument to easily find conflict
2. Define the value of optimal solution(state) recursively:

   2.1 Let dp(i) denote the maximum value got with a rod of length i

   2.2 $dp(i)$ is initialized to 0 for all $1 \leq i \leq n$

   2.2 $dp(1) = p_1$

   2.3 $dp(n) = \max_{n_l, n_r : n_l + n_r = n} \{dp(n_l) + dp(n_r)\}$

The left two step is to convert the recursion into interative version

Some more optimization:

- Use symmetric to reduce the number of comparision needed
- The left most of cuts must be one of P hence $dy(n) = \max\{p(n) + p(1) + dy(n-1) + \ldots + p(1)\}$, this optimization is especially greate when the number of prices are significant less than n(i.e. $P = (p_1, p_2, \ldots, p_k), k << n$)

Longest increasing subsequence problme(LIS)

Given a sequence of numbers $A = (a_1, a_2, \ldots, a_n)$, find $(k_1, k_2, \ldots, k_t)$ with the maximum value t, with $k_1 < k_2 < \ldots < k_t$ such that $A(k_i) \leq A(k_j)$ given $i \leq j$

Method 1:

- dp[i]: the maximum length of increasing subsequence
- dp[i] = 1 initially, for $i \in \{1, 2, \ldots, n\}$ and 0 otherwise
- dp[i] = $\max_{j \leq i \wedge A[[j] \leq A[i]]} dp[j] + 1$
- Result $\max_{i \in \{1, \ldots, n\}} \{dp[i]\}$

Method 2:

- dp[i]: the index of minimum element e, which which there subsequence of length i ending with e
- len: the current length of maximum subsequence
- dp[i] =∞ initially
- dp[len+1] = $dp[i]$ if $A[i] \geq dp[len]$
- dp[j] = i if $A[i] \leq A[dp[j]] \wedge A[i] \geq A[dp[j-1]]$ for $j = 1$ to len

Longest Palindromic Substring

$dp[i][j]$: 1 if string[i:j] is palindromic; 0 otherwise

$dp[i][i] = 1$

$dp[i][i+1] = 1$ if string[i] == string[i+1] for i=1 to n-1

result: $max\{|i-j| : dp[i][j] == true\}$

Some other example:

- Applications:
  - Content-Aware Image Resizing (Graphics)
  - All-Pairs Shortest Paths (Routing)
  - Edit Distance (Auto-correct, DNA similarity)
  - Longest Common Subsequence (`diff` / `git diff`)
  - Parsing (See CYK algorithm)
  - Query Optimization (Databases)

Lecture 9: Greedy algorithm

General idea:

Recast the problem so that only one subproblem should be solved each one step. Moreover, the program can make greedy choice given the result of the subproblem to achieve optimize result to the original problem.

Proof of correctness of greedy algorithm

- Optimal substructure
  - Usually by cut and paste argument
- Greedy choice
  - There is at least one optimal selection with the choice; Because each step has only one subproblem

The difference between greedy algorithm and dynamic programming:

- Greedy algorithm is like a list, where the next result depends only on current output, henc the greedy choice of each step will finally get an optimal final result.
- Dynamic programming is like a tree, where each nodes depends on all its child, hence the optimal of each nodes will not necessarily yield the optimum result of child node by game theorem.

Concrete example:

Fractional Knapsack

If the objects are not atomic but some dividable products(such as water and oil). Then given the instance $(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and W; optimum of $\sum_i v_i \cdot \frac{x_i}{w_i}$ would optimize the overall value(because the W is fixed and no blank space left and a higher dencentration means higher quality)

```
FK(((w1, v1),(w2,v2),...,(wn,vn)),W):
  valperkg <- [1...n]
  sort valperk, where i<=j iff vi/wi <= wj/wj
  for i<- n to 1
      if W == 0:
        break
      j < valperk[i]
      w_tmp <- min(W,w[j])
      print(w_tmp+" kgs of item "+ j)
      W <- W-k
```

Proof of correctness

RTBD

Minimum Spanning Tree(MST)

Given a graph G = (V,E), and a mapping $w : E \to R^+$, find a minimum tree containing all $v \in V$ with minimum weights.

Proof of optimal substructute

1. Given a MST $T = (V_T, E_T)$ of graph G, $\forall (u, v) \in E_T$: say $T = T_1 \cup \{(u, v)\} \cup T_2$, where $T_1 = (V_{T_1}, E_{T_1}), T_2 = (V_{T_2}, E_{T_2})$ and $u \in V_{T_1}, v \in V_{T_2}$
2. Then T_1 is MST of $G \backslash \{V_{T_2}, E_2\}$, where $E_2 = \{(u, v) \in E : u, v \in V_{T_2}\}$
3. Otherwise if there is another spanning tree T' with lower weights, then by cut and paste argument

$T_1' \cup \{u, v\} \cup T_2$ will be a ST with lower weights thant T, which conflicts that T is MST.

4. Hence optimal substructure proved

Proof of greedy choice

1. Let $(u, v) \in E$ be the edge with minimum wightage, there is a MST of G that contains (u,v)
2. Assume there is a MST that does not contain (u,v). There is a path from u to v, remove any edge on the path and add (u,v) will keep the new tree
    - The new tree a spanning tree because each nodes have degree 2 on the path and removal of one path will not decrease any node to degree 1.
    - The new tree will have weight no less than the original MST because weight of (u,v) is less or equal to the weight of edge removed.
3. Then a MST containing (u,v) is made and the greedy choice is proved

Prim's algorithm

```
Prim(V,E):
  MinimumHeap mh;
  T_E = {}
  T_V = {}
  v <- V[0]
  V <- V\v
  T_V.push(v)
  for ((v,u) in E):
    mh.push(((v,u),u))
    E <- (v,u)
  while V not empty:
    while True:
      e, v <- mh.head()
      mh.pop()
      if v not in T_V:
        T_V.push(v)
        for ((v,u) in E):
          mh.push(((v,u),u))
          E <- (v,u)
```

Haffman Code

RTBD

Lecture 10: Reduction and Intractability

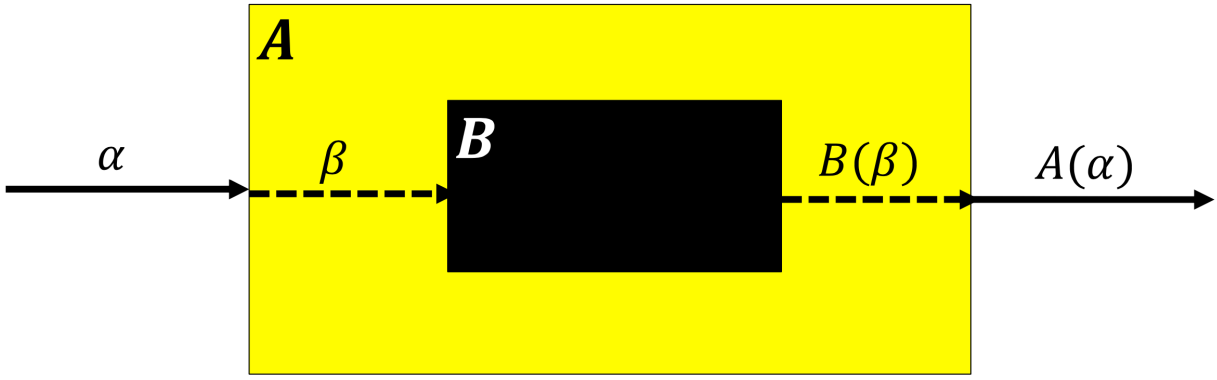
Instance

An instance of an algorithm is all input needed to compute the result of hte algorithm

Reduction

A way to solve problem. Generally format:

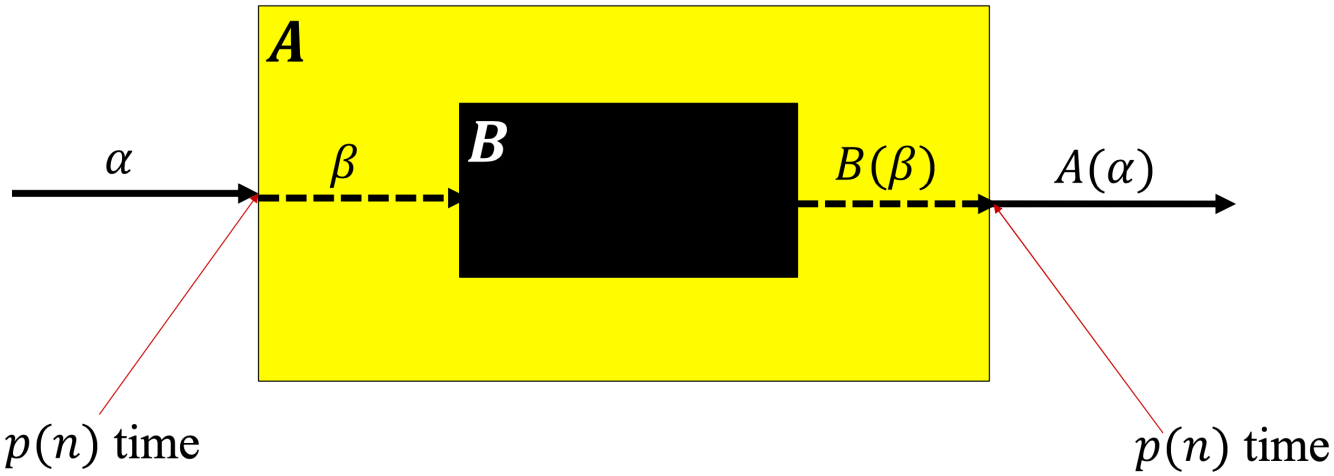
1. Given a algorithm A, convert instance of A to instance of B
2. Execute algorithm B to get the result
3. Convert the result of B to result of A



p(n)-time Reduction

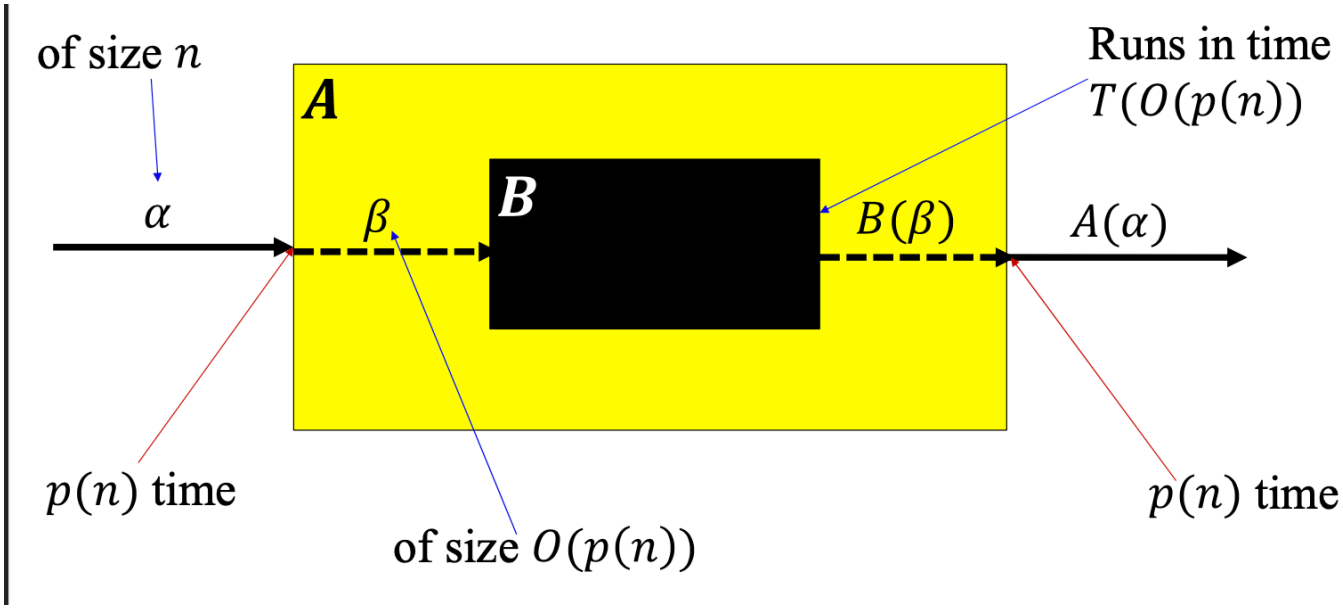
- A instance β of problem B can be converted from instance α of problem A
- A solution of problem A can be recovered from solution of problem B

We say that there is a p(n)-time reduction from A to B



Time complexity of p(n)-time reduced algorithm

If there is a p(n)-time reduction from problem A to B, and there is a T(n)-time algorithm to solve problem B on instance of size n. Then there is an T(O(p(n)))+O(p(n)) algorithm to solve problem A with instance of size n.



Polynomial Time Reduction

If there is a p(n)-time reduction from A to B, where $p(n) = O(n^c)$ for some constant c, then we say that there is a polynomial time reduction from A to B, denoted as $A \leq_p B$

Notably, if $A \leq_p B$, and there is an polynomial algorithm to B, then there is a polynomial algorithm to A.

Polynomial

When talking about the "polynomial" of an algorithm, we mean that the algorithm is polynomial in the length of encoding of the instance.

Standard encoding:

- Integer: binary encoding
- Mathematical objects(Graph, Array, Matrix): list of data

Pseudo-polynomial

An algorithm runs in polynomial to numeric value of the input of instance but in exponential to the length of input is called pseudo-polynomial.

Concrete example

Matrix Squaring and Matrix Multiplication

- MATRIX SQUARING

  Given a $n \times n$ matrix A, calculate $A \times A$

- MATRIX MULTIPLICATION

  Give two $n \times n$ matrices A,B, calculate $A \times B$

Reduce MATRIX SQUARING to MATRIX MULTIPLICATION is obvious

Reduce MATRIX MULTIPLICATION to MATRIX SQUARING:

1. Given matrices A, B, construct matrix

$$\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}$$

0-Sum and T-Sum

- 0-sum:

  Given a set of numbers $A = \{a_1, a_2, \ldots, a_n\}$, decides whether there are two distinct elements $a_i, a_j \in A$ such that $a_i + a_j = 0$

- T-sum

  Given a set of numbers $A = \{a_1, a_2, \ldots, a_n\}$ and T, decides whether there are two distinct elements $a_i, a_j \in A$ such that $a_i + a_j = T$

Reduce T-sum to 0-sum:

1. Given instance of T-sum $A = \{a_1, a_2, \ldots, a_n\}$ and T, make $A' = \{a_i - T/2 | a_i \in A\}$
2. A' as instance of 0-sum

Intractability

In computational theory, a problem is intractable if there is no efficient algorithm to solve them.

Decision Problem vs Optimization Problem

Decision Problem

Decision algorithm is a function that maps the space of instance to $\{True, False\}$

Optimization Problem

An optimization problem:

- An predefined totally ordered set of the result of algorithm S
- An adversary that tries to compromise the algorithm

An optimization algorithm is a function that maps the space of instance to space of result of the problem such that no super-polynomial adversary can compromise the result of the algorithm by providing better ordered results.

Reduction From Decision Problem to Optimization Problem

Instance of decision problem is usaully the instance of optimization problem I plus a element k of the pre-defined totally ordered set S.

Hence given stantance of decision problem (I, k), setting (I) as the instance of optimization problem and executes Optimization Problem. The result of Decision Problem can be recovered by comparing k with the output of Optimization Problem $v_{optimized}$:

- If $k \leq v_{optimized}$: return True
- else return False

Reduction From Decision Problem To Decision Problem

Given two decision problem A,B, we can reduce A to B following the format:

1. Show that there is a polynomial $p(n)$ converting instance α of A to instance β of B
2. Shows that α is an YES instance of A $\rightarrow$ β is an YES instance of B
3. Shows that β is an YES instance of B $\rightarrow$ α is an YES instance of A

Concrete Example

Vertex Cover

Given a graph G=(V,E), is there a subset $V' \subseteq V$ with size k(or fewer) such that each edge $e \in E$ is incident to at least one vertex of V

Independent Set

Given a graph G(V,E), is there a subset $V' \subseteq V$ such that $\forall u, v \in V' : (u, v) \notin E$

Set Cover

Given a set of integers S={1,..,n}, is there $\leq k$ subsets of S whose union equals to S

Satifiability:

- Literal: A boolean variable or its negation
- Clause: A disjunction(OR) of literals
- Conjective Normal Form(CNF): a formula ϕ that is conjunction(AND) of clauses.

NP-Complete

NP problem

NP problem is the class of problems that have polynomial verifiable certificate for YES instance

$$P \subseteq NP$$

It's obvious that P problem has a polynomial verifiable certificate, because the verification algorithm should be no harder than the solution algorithm (Otherwise it's trival to claim that it's algorithm solving the problem). On the other hand, there must exist a verification algorithm that solves the P problem in polynomial time and verify itself

NP-hard

A problem A is said to be NP hard if $\forall$ Problme $B \in NP : B \leq_P A$

NP-Complete

A problem A is said to be NP-Complete if it's both NP and NP-hard.

Cook-Levin Theorem

Any problem in NP reduced to 3-SAT

Tips of reduction:

- try something trivial, and see how it fails.

Concrete Example

Reduce 3-SAT to independent problem

refers to NP-complete note

Ham-cycle:

Given a graph G=(V,E), the Ham-cycle problem is to decide whether there is a **simple** cycle that passes every vertex $v \in V$

Interesting time complexity

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

Firstly, the master's method cannot be applied here:

1. Let $n = 2^m, T(2^m) = 4T(2^{m-1}) + 2^{2m} \lg m$
2. Let $G(m) = T(2^m) : G(m) = 4G(m-1) + 4^m \lg m = 4^m \cdot \sum_{i=1}^{m} \lg i = 4^m \lg m! = \Theta(4^m \cdot m \lg m)$
3. Because $m = \lg n$, $G(m) = \Theta(n^2 \lg n \lg \lg n)$

$2^n << (\lg n)^n << n! << n^n$

Useful knowledge

- $\sum_{i=1}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{i} di = \ln i$
- $\sum_{i=1}^{n} \frac{1}{i^2} \leq \sum_{i=1}^{n} \frac{1}{i*(i-1)} = 1 - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} \cdots + \frac{1}{n-1} - \frac{1}{n} = \frac{n-1}{n}$