

Algorithms

R4 Cheng

March 6, 2025

Def. Introduction of precise, unambiguous, and correct procedures for solving general problems

We care how many clicks in debug mode

Runtime Analysis

Runtime is expressed by counting the number of steps, as function of **the size of the input**

Asymptotic Notations:

- Big O : upper bound on the growth rate
- Little o : "strict" upper bound on the growth rate
- Big Ω : lower bound on the growth rate
- Θ : tight bound on the growth rate

$f(n)$: the runtime of an algorithm for input size n

$g(n)$: a simplified function without constants, lower order terms, e.g. n^2 , $n \log n$

Big-O Definition: It is said $f(n) = O(g(n)) \iff$ there exists a constant $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Little-o Definition:

$$f(n) = o(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

E.g. $f(n) = 3900n + 57800$

$f(n) = o(n^2)$? T, because $\lim_{n \rightarrow \infty} \frac{3900n+57800}{n^2} = 0$

$f(n) = o(n)$? F, because $\lim_{n \rightarrow \infty} \frac{3900n+57800}{n} = 3900$

Big-Ω Definition: It is said $f(n) = \Omega(g(n)) \iff$ there exists a constant $c > 0$ and $n_0 > 0$ such that

$$f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0$$

Relation between O, o, Ω, θ

- $f = \theta(g) \iff f = O(g), f = \Omega(g)$
- $f = o(g) \Rightarrow f = O(g)$
- $f = O(g) \iff g = \Omega(f)$
- $f = \theta(g) \iff g = O(f)$

Caveats about constants

For $\log_4(n)$, is 4 ignorable? No, it matters

$$> \log_a(x) = \log_a b \log_b(x)$$

Common Efficiency Class

Class	Name	
1	constant	No reasonable examples, most cases infinite input size requires infinite run time
$\log n$	Logarithmic	Each operation reduces the problem size by half, Must not look at the whole input, or a fraction of the input, otherwise will be linear
n	linear	Algorithms that scans a list of n items (sequential search)
$n \log n$	linearithmic	Many D&C algorithms e.g., merge sort
n^2	quadratic	Double embedded loops, insertion sort
n^3	cubic	Three embedded loops, some linear algebra algo.
2^n	exponential	Typical for algo that generates all subsets of a n element set.
$n!$	factorial	Typical for algo that generates all permutations of a n-element set

Merge Sort

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n + c$$

> $2n$: compare and copy n elements = $2n$

Proof

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + 2n + c \\ \Rightarrow T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{4}\right) + (n + c) \\ \Rightarrow T\left(\frac{n}{4}\right) &= 2T\left(\frac{n}{8}\right) + \left(\frac{n}{2} + c\right)\end{aligned}$$

Bring in $T\left(\frac{n}{2}\right)$

$$\begin{aligned}\Rightarrow T(n) &= 2\{2T\left(\frac{n}{4}\right) + (n + c)\} + 2n + c \\ &= 2^2T\left(\frac{n}{2^2}\right) + 2 \cdot 2n + c + 2c\end{aligned}$$

Bring in $T\left(\frac{n}{4}\right)$

$$\begin{aligned}&= 2^2\{2T\left(\frac{n}{8}\right) + \left(\frac{n}{2} + c\right)\} + 2 \cdot 2n + c + 2c \\ &= 2^3T\left(\frac{n}{2^3}\right) + 3 \cdot 2n + c + 2c + 4c \\ \Rightarrow T(n) &= 2^kT\left(\frac{n}{2^k}\right) + k \cdot 2n + (2^k - 1)c\end{aligned}$$

Since $T(1) = 1$, $\frac{n}{2^k} = 1 \Rightarrow k = \log_2(n)$

$$\begin{aligned}\Rightarrow T(n) &= n \cdot 1 + \log_2(n) \cdot 2n + (n - 1)c \\ \Rightarrow T(n) &= O(n \log n)\end{aligned}$$

Master Theorem

$$\begin{aligned}T(n) &= aT\left(\frac{n}{b}\right) + cn^d \\ \Rightarrow T(n) &= \left(1 + \frac{a}{b^d} + \dots + \frac{a^k}{b^{kd}}\right)cn^d\end{aligned}$$

a : number of subproblems

b : factor by which the problem size is reduced

d : exponent in the running time of the "combine" step

Case 1: $a < b^d$, or equiv. $d > \log_b a$, $T(n) = O(n^d) \Rightarrow$ the cost of the "combine" step dominates the cost of the "split" step

Case 2: $a = b^d$, or equiv. $d = \log_b a$, $T(n) = O(n^d \log n)$

Case 3: $a > b^d$, or equiv. $d < \log_b a$, $T(n) = O(n^{\log_b a})$

Proof by induction

> Useful for proving correctness of recursive algorithms

Devide and Conquer

Inversion Counting

Input: an A containing numbers $1, 2, \dots, n$ in some arbitrary order

Output: the number of inversions in A , i.e. the number of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$

It is useful for recommender systems \Rightarrow fewer inversions means better alignment with user preferences

Median

Input: A list of numbers S ; an integer k

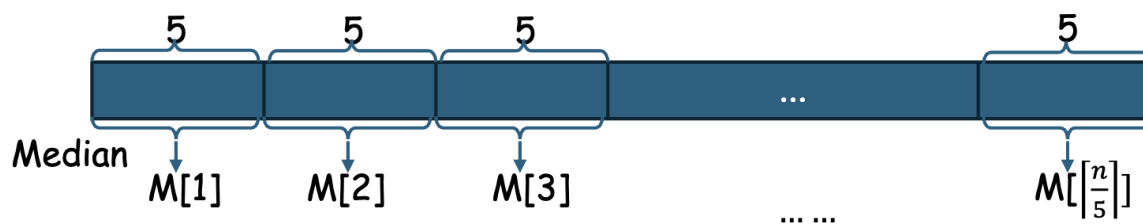
Output: the k th smallest element of S

0.0.1 A randomized divide-and-conquer algorithm

$$S_k =$$

Smart Selection of Pivot

Break array A into chunks of fives and find the median of each chunk, constructing a new array M of medians.



And **median** of M aka. MM is the pivot.

Claim: MM is at least $\geq \frac{3}{10}A$ and $\leq \frac{3}{10}A$

Proof: MM is the median of medians, so $\frac{1}{2}$ medians in M are $\leq MM$. $\therefore M = \frac{A}{5} \Rightarrow \frac{A}{10}$ medians are $\leq MM$. Moreover, For each $M[i]$, there are 3 elements $\leq M[i] \therefore \frac{3A}{10} \leq MM$

Select(A , k)

1. for $i = 1, \dots, n/5$
2. $M[i] = \text{median}(a[5(i-1)+1], 5i)$
3. $MM = \text{Select}(M, n/10)$
4. $v = \text{index of } MM \text{ in } A$

```

5. r = partition(A, v)
6. if k=r: return A[r]
7. if k<r: Select(A[1,...,r-1], k)
8. if k>r: Select(A[r+1,...,n], k-r)

```

$$T(n) \leq O(n) + T(n/5) + T(7n/10) \Rightarrow T(n) = O(n)$$

Dynamic Programming

The main question in dynamic programming always is, what are the subproblems?

Fibonacci

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\Rightarrow T(n) \leq 2T(n-1) + c \\
&\Rightarrow T(n) \leq 2^2T(n-2) + 2c + c \\
&\Rightarrow T(n) \leq 2^kT(n-k) + kc \\
k = n-1 &\Rightarrow T(n) \leq O(2^n)
\end{aligned}$$

$$T(n) \geq 2T(n-2) + c = \Omega(2^{\frac{n}{2}})$$

Applications

- Unix diff for comparing files
- Bellman-Ford for shortest path
- CKY algorithm for natural language parsing
- etc.

Longest Increasing Subsequence

$L(i)$ = the length of a longest increasing subsequence ending at position i

$$L(i) = (\max_{j: a_j < a_i} L(j)) + 1$$

Edit Distance

$D(i, j)$ = the edit distance between s_1, s_2, \dots, s_i and t_1, t_2, \dots, t_j

Maximum Subarray

Because the decision at each index depends only on the previous index, it is unnecessary to define $L(i, j)$ = the maximum subarray sum from index i to j . Instead, we can define $L(i)$ = the maximum subarray sum ending at index i .

Knapsack

Q: For max W pounds, there are n items to pick from, of weight w_1, \dots, w_n and dollar value v_1, \dots, v_n . What is the most valuable combination of items he can fit into his bag?

E.g., take $W = 10$ and

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

0-1 knapsack problem

Intuition: Go greedy

1. Greedily choose the most valuable item?
2. Greedily choose the most value/weight item?

Sadly, both can be found a counter example.

$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, 2, \dots, j$

We should consider both capacity w and items j , where $0 \leq j \leq n$

\Rightarrow

The answer we seek is $K(W, n)$

Subproblems:

1. Choose the current item: $K(w - w_j, j - 1) + v_j$
2. Don't choose the current item: $K(w, j - 1)$

$$\Rightarrow K(w, j) = \max \begin{cases} K(w - w_j, j - 1) + v_j \\ K(w, j - 1) \end{cases}$$

(The first case is invoked only if $w_j \leq w$)

> we can express $K(w, j)$ in terms of subproblems $K(\cdot, j - 1)$.

Time complexity: $O(nW)$

Unbounded knapsack problem

> aka. [Knapsack with repetition](#)

$K(w)$ = maximum value achievable with a knapsack of capacity w

Express this in terms of smaller subproblems:

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

$K(0) = 0$

for $w = 1$ to W :

$K(w) = \max \{K(w - w_i) + v_i : w_i \leq w\}$

return $K(W)$

It is 1D dynamic programming. Each entry can take up to $O(n)$ time to compute, so the overall runtime is $O(nW)$

Optimal binary search tree

Q: what is the time complexity for this q?

TODO: study Longest common subsequence

Longest Common Subsequence

Greedy Algorithms

Greedy template:

Let T be the set of tasks H is the rule determining the greedy algorithm

$A = \{\}$

While (T is not empty)

 Select a task t from T by a rule H

 Add t to A

 Remove t and all tasks incompatible with t from T

Proving the optimality of a greedy algorithm

Greedy stays ahead

Exchange argument

Scheduling Theory

0.0.2 Interval Scheduling

$f()$

Prove:

the **Earliest-Finish-Time** algorithm is optimal.

We need to show that the solution chosen by greedy is as good as any other solution.

Usually, use the strategy of proof by contradiction

Minimum Spanning Tree

> aka. every node should be connected to every other node

We assume the graph is undirected in this class.

0.0.3 Applications of MST

- Network design
- Approximations to NP-hard problems like traveling salesman problem
- Clustering in machine learning

0.0.4 Greedy Template for MST

Init an empty set of edges T

While T is not yet a spanning tree ($|T| \leq |V| - 1$)

 select $e \in E$ to add to T according to a greedy criterion

Return T

Kruskal's Algorithm: sort edges by weight, add edge to T if it doesn't create a cycle

Prim's Algorithm: start at any node, add the lightest edge to T that connects a node in T to a node not in T

0.0.5 Proof of Correctness

Use **cut property**: If we can partition the graph into two parts, and an edge e is the cheapest edge connecting across the two parts, then e must be in the MST.

Huffman coding

Claim: The optimal code must be a full binary tree

// C : the set of tokens, f : the frequency of each token

Huffman_recur(C , f)

 Identify the two least frequent tokens x , y in C

$C' = C - \{x, y\} + \{z\}$ // $z = x + y$

 Assign frequency for z : $f(z) = f(x) + f(y)$

$T' = \text{Huffman_recur}(C', f)$

 Append x and y as children of z in T'

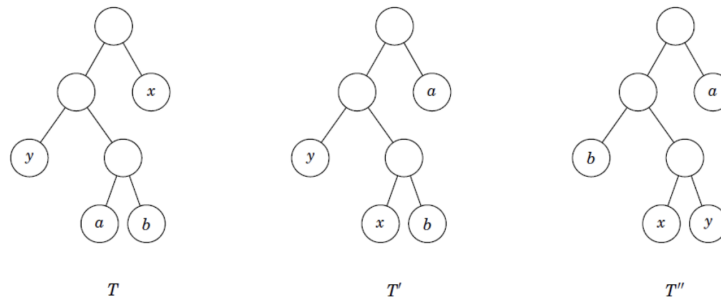
 return T'

0.0.6 Proof Greedy Choice Property

Let x, y be the two least frequent tokens of C . There exists an optimal tree in which x and y are siblings.

⇒ use **the exchange argument**

- Let T be an optimal tree where x and y are not siblings.



- Identify the deepest sibling pair in T , let's say they are a and b
- We can replace a with x , b with y , which will not increase the cost

Proof Correctness of Huffman's Algorithm

Proof by induction

Base case: $n = 2$ and 1 bit for each token

Inductive hypothesis:

Assume that the algorithm produce an optimal code for all token of $2, \dots, k$

Inductive step:

Linear Programming

- least squares
- minimize the maximum absolute deviation $\max_{1 \leq i \leq n} |y_i - (ax_i + b)|$

min $-x_1 - x_2$

1. $4x_1 - x_2 + s_1 = 8$

2. $2x_1 + x_2 + s_2 = 10$

3. $5x_1 - 2x_2 - s_3 = -2$

4. $x_1, x_2, s_1, s_2, s_3 \geq 0$

Problem: Maximum Flow

target: $\max f_{sa} + f_{sb} + f_{sc} = \max f_{dt} + f_{et}$

Reduction

0.1 Satisfiability

Literal: x_i or $\neg x_i$

Clause: A disjunction of clauses $c - j = x_1 \vee x_2 \vee \neg x_3$

CNF ϕ