Data Structures and Algorithms II

Nate Annau

Table of Contents

1. Greedy Algorithms

1.1. Introduction

Lecture 1 Jan 7

1.1.1. Concept

There's no exact definition of a greedy algorithm, but the general idea is that we always take the most optimal local step.

1.1.2. Computational Problem

If the number of quarters in some change is a, the number of dimes is b, the number of nickels is c, and pennies d, then we want to choose a, b, c, d such that

1. a, b, c, d nonnegative integers
2. $X = 25a + 10b + 5c + d$
3. Minimize $a + b + c + d$

The coin change problem asks for how to solve this optimally.

1.1.3. Algorithm

We use as many quarters as possible, then use as many dimes as possible and so on.

1. Choose 2 quarters, leaving the remainder as $73 - 2 \times 25 = 23$
2. Choose 2 dimes with remainder $23 - 2 \times 10 = 3$
3. Choose 0 nickels
4. Choose 3 pennies, with remainder 0.
5. Solution is $a = 2, b = 2, c = 0, d = 3$

1.1.4. Remark

Consider a nation called Zenobia where they have coins with value 12 and pennies and nickels as normal. Notice a greedy algorithm here for $X = 15$ gives us the solution $15 = 12 + 1 + 1 + 1$, which is not optimal because we have the shorter decomposition $15 = 5 + 5 + 5$. This shows that the optimality of the greedy algorithm for the coin change problem is not clear.

1.1.5. Exercise 1

> Prove that the greedy algorithm for the coin change problem is optimal.

Solution: Proof:

□

1.1.6. Definition

We are given an undirected graph $G = (V, E)$. A **vertex cover** of G is a subset $C \subseteq V$ of vertices so that every edge of E has at least one of its endpoints in C.

1.1.7. Computational Problem

Our problem is to find the smallest size vertex cover of a graph G.

Greedy Algorithm: pick the vertex u of maximum degree. Remove u and all of its neighbors and add it to the cover. Repeat until there are no more vertices.

1.1.8. Definition

Given a graph $G = (V, E)$ with n vertices, we want to assign colors $\{1, 2, ..., k\}$ such that no edge gets the same color at both endpoints.

1.1.9. Computational Problem

Our problem is to color a given graph with a minimum number of colors.

Greedy Algorithm: Consider the vertices in some order, i.e., $v_1, ..., v_n$. Now assign v_i the lowest available color not used by v_i's neighbors among $\{v_1, ..., v_{i-1}\}$, adding a fresh color if needed.

1.1.10. Computational Problem

Suppose a trucker is driving from city A to city B along a route map with gas stations along the way. The trucker would like to make the *fewest* possible stops along the way. Problem: Given any such route and locations of gas stations, plan the trucker's stops.

Greedy Algorithm: If he has some max distance R he can go, always take the gas station furthest along with distance less than R.

1.1.11. Computational Problem

A salesman needs to visit n cities, say $x_1, x_2, ..., x_n$. The salesman knows the travel cost between every pair of cities. Starting at x_1, the salesman must visit all other cities before returning to x_1, at minimum total cost. In which order should he visit?

Greedy Algorithm: Always travel to the nearest city that hasn't been visited.

1.1.12. Remark

It turns out that only the Trucker's Problem Greedy Algorithm is right – the other algorithms are not optimal.

1.2. Interval Scheduling

Lecture 2 Jan 9

1.2.1. Computational Problem

1. Suppose we have a list of n activities that we want to schedule on a single resource (processor).
2. Each activity is specified by its start and end times.
3. Only one activity can be scheduled on the resource at a time.
4. Each activity uses the resource continuously between its start and end times.

Our goal is to schedule the maximum possible number of activities.

1.2.2. Example

- Suppose we have 5 activities:

$$\{(3,6),(1,4),(4,10),(6,8),(0,2)\}$$

  - these are like intervals on the number line
- A feasible schedule cannot have two activities that overlap (in time).
- For instance, we cannot accept both (1, 4) and (3, 6).
- However, (3, 6) and (6, 8) are acceptable, because second only begins when first ends.
- The optimal solution in this exmaple has size 3.

1.2.3. Remark

We can formalize the problem as followed.

- We have a set of activities, which we denote by $S = \{1, 2, ..., n\}$
- The ith activity is specificed by a tuple $(s(i), f(i))$ with $s(i) \leq f(i)$, for $i = 1, ..., n$
- A *feasible schedule* is a subset in which no two activities overlap
- Goal: find a feasible schedule of maximum size.

Notice a naive algorithm searches through all 2^n subsets, and outputs the largest feasible one, but this is not efficient.

1.2.4. Remark

We have a few possible greedy strategies:

1. FIFO: pick the one that starts first, remove overlapping activities and repeat
2. Shortest first: pick the activity with the shortest duration, remove overlapping activities and repeat
3. Min overlap first: Count the number of other jobs that overlap, then choose one with smallest overlap count

But if we analyze these, we see simple counterexamples for the first one (an activity that's really long is first) and the third one (a shorter interval in the middle that overlaps with 2 which could fit together). We could also construct an example for the second.

1.2.5. Algorithm

The correct greedy strategy is to sort the process job in order by earliest finish time.

We sort jobs by order:

$$f(j_1) \leq f(j_2) \leq \cdots \leq f(j_n)$$

Pseudocode:

```
A = {1}; j = 1; // accept job 1
for i = 2 to n do
  if s(i) >= f(j) then
    A = A + {i}; j = i;
return A
```

1.2.6. Example

Activity	Start	Finish
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

- The greedy algorithm first chooses 1, then skips 2 and 3
- Next it chooses 4, and skips 5, 6, and 7
- It should choose 8 and 11 (verify)

1.2.7. Remark

- Suppose that OPT is an optimal solution for the problem. Ideally, we would like to show our algorithm always returns the same output, namely $A \equiv \text{OPT}$.
- But there may be multiple optima, and the best we can hope for is that $|A| = |\text{OPT}|$
- The proof idea (typical for optimality of greedy algorithms) is to show the greedy algorithm stays ahead of the optimal solution at all times

1.2.8. ***Proposition***

The Earliest Finish Time Algorithm correctly solves the inveral scheduling problem.

Proof: Suppose $a_1, \ldots, a_k$ are the indices of jobs in the greedy schedule, and $b_1, \ldots, b_m$ are jobs in an optimal schedule OPT. We want to show $k = m$. Mnemonically, a_i's are jobs picked by our algorithm while b_i's are the optimal schedule jobs. (Note we list both in order of start times.)

- Our intuition should be that the greedy algorithm makes the resource free again as soon as possible. In particular, $f(a_1) \leq f(b_1)$, that is, the greedy algorithm stays ahead. We formalize this as follows.

1.2.9. ***Lemma***

For every $i \leq k$, we have $f(a_i) \leq f(b_i)$.

Proof: We proceed by induction. Note the $i = 1$ case is trivial. By the induction hypothesis, we have $f(a_{i-1}) \leq f(b_{i-1})$.

Notice $f(b_{i-1}) \leq s(b_i) \leq f(b_i)$ and $f(a_{i-1}) \leq s(a_i) \leq f(a_i)$.

Therefore $f(a_{i-1}) \leq s(b_i)$, since clearly $f(b_{i-1}) \leq s(b_i)$. But notice $s(b_i) \leq f(b_i)$, which implies $f(a_i) \leq f(b_i)$ as desired.

□

Notice by the lemma, $f(a_k) \leq f(b_k)$, but $f(b_k) \leq f(b_m)$, so $f(a_k) \leq f(b_m)$, implying $k \leq m$. But $m \leq k$ since OPT is optimal, implying $m = k$ as desired.

□

1.2.10. Runtime Analysis

The runtime of the previous algorithm is $O(n \log n) + O(n) = O(n \log n)$ because of the sorting and then iteration through.

1.2.11. Remark

Consider the same setup with the following variant. Given a set of activities, what is the smallest number of machines needed to schedule them all?

A natural greedy algorithm to try is the following: use EFT to find the maximum number of activities that can be scheduled on one machine. Delete those and repeat the rest until no activities are left.

However, this algorithm fails because we can construct a setup where EFT uses 3 machines but the optimal solution is 2 machines.

Instead, a simpler greedy algorithm works:

```
Sort activities by start time
Put activity 1 on machine 1
for i = 2 to n:
  if i can be scheduled on any of the existing machines, add to that machine otherwise schedule
activity i on a new machine
```

1.2.12. ***Proposition***

The previous algorithm is optimal.

Proof: Define the depth d as the maximum number of activities working at the same time. Notice that OPT >= depth, because we have to run all processes. Further, Greedy <= depth (since we only start on a new machine when there are $d - 1$ processes running). Therefore, the Greedy algorithm has to be optimal.

□

1.3. Huffman Code

Lecture 3 Jan 14

1.3.1. Example

1. Suppose we have a signal digitized at some sampling rate, say 44 KHz.
2. This produces a sequence of real numbers $s_1, s_2, \ldots, s_T$
3. Each s_i is quantized - for example, to 256 different values
4. The quantized string T over alphabet G is encoded in binary
5. This last step uses Huffan encoding - we want to write this as compactly as possible

1.3.2. Example

Consider a data file with 100K characters. THe file contains only 6 different characters with the following frequency distribution

Char	a	b	c	d	e	f
Freq(K)	45	13	12	16	9	5

Suppose the cost of storage or transmission is proportional to the number of bits.

1.3.3. Computational Problem

We want to design binary codes to achieve maximum compression. With 6 characters, we need at least 3 bits to represent each. One possible set of such codes is

Char	a	b	c	d	e	f
Code	000	001	010	011	100	101

How can we reduce the storage as much as possible?

1.3.4. Concept

The previous code used fixed length coding. What if we instead use shorter codes for more frequent letters? One possible set of variable length codes is the following:

Char	a	b	c	d	e	f
VLC	0	101	100	111	1101	1100

With this coding scheme, we only need 224 Kbits:

$$1 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5$$

This is a 25% improvement over fixed length codes. In general, they can give huge savings.

1.3.5. Concept

- We have a potential problem with variable length codes–how do we know where the boundaries are?
- For example if 0 encodes x and 00 encodes y, how do we interpret 000?
- We could put special markers between letters but that loses efficiency
- Instead we will ensure our codes satisfy the following property: *no codeword can be a prefix of another code*
- {0, 000} are invalid prefix codes, but {0, 101, 100, 111, 1101, 1100} are valid
- To encode, just concatenate the codes for each letter of the file; to decode, extract the first valid codeword and repeat
- Example: code for 'abc' is 0101100, and 001011101 uniquely decodes to 'aabe'

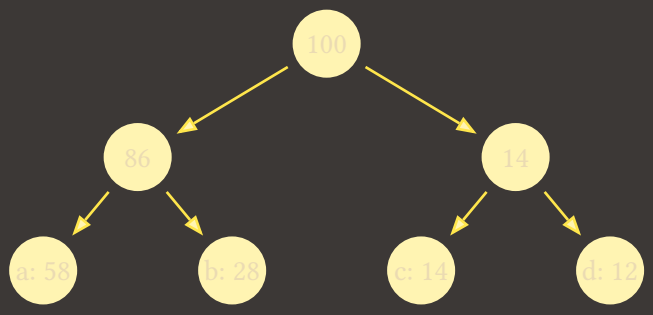
1.3.6. Concept

We represent codes using a tree as follows:

- Code for a letter is the sequence of bits between root and that leaf
- Decoding algorithm: start at root and output leaf
- Prefix code: only leaf nodes correspond to codes, not internal nodes

Note that the constant length coding is a balanced binary tree where we always insert in the next position, whereas variable length coding is not necessarily balanced.

Note that if a node has children, it cannot have an associated letter, because it will be a prefix of the children.



1.3.7. Remark

Observe that the tree for an optimal code must be full: that is, each internal node has two children. Otherwise, we can improve the code. Thus the fixed length code cannot be optimal!

1.3.8. Computational Problem

Let C denote our alphabet (character set) and $f(p)$ the frequency of a letter p. Let T be the tree for a prefix code, and $d_T(p)$ th depth of p in T. The total number of bits (bit complexity) needed to encode our file using this code is:

$$B(T) = \sum_{p \in C} f(p) d_T(p)$$

We want a code that achieves the minimum possible value of $B(T)$.

1.3.9. Algorithm

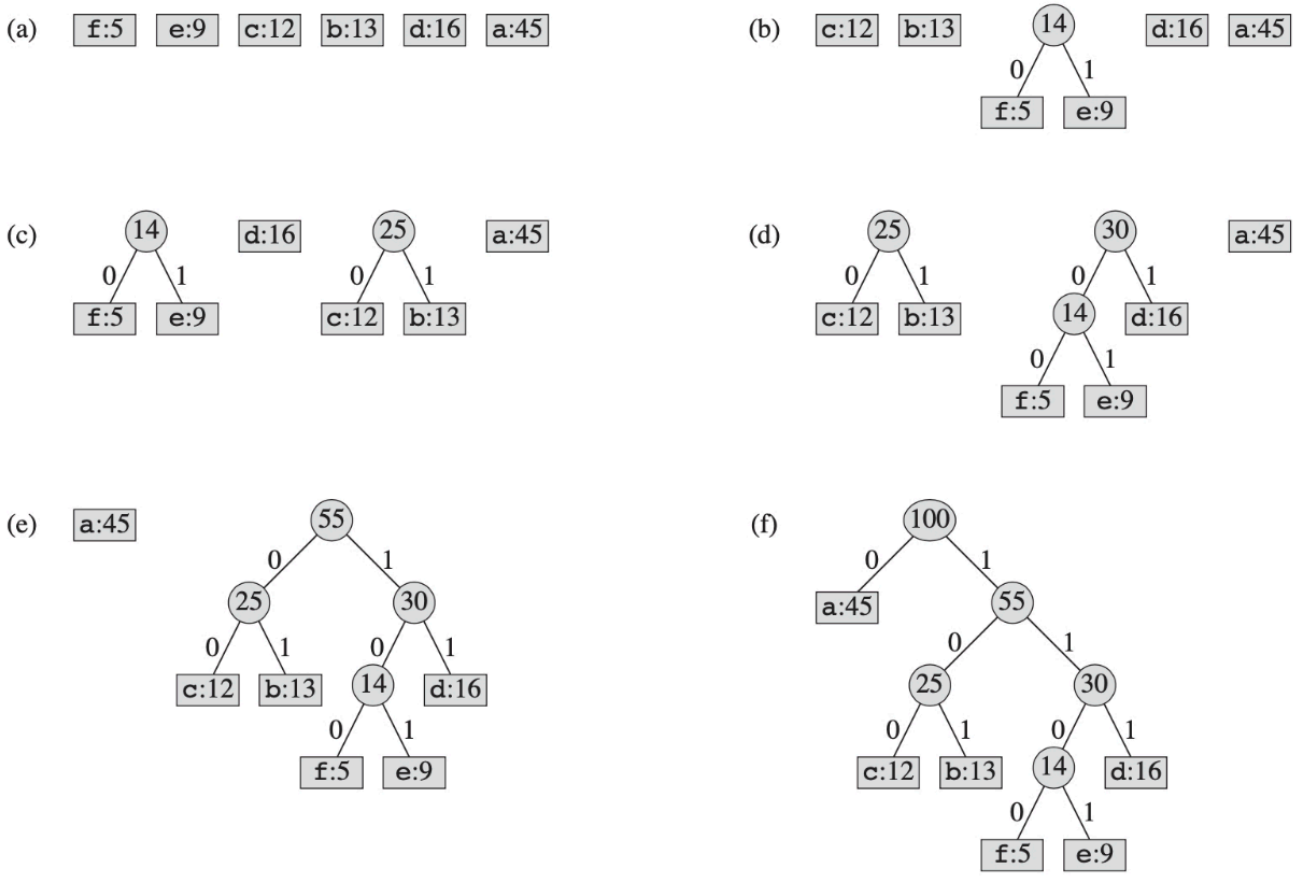
We can intuitively think of this as building the best tree T to represent binary codes.

Iniitally, each letter is represented by a single node tree, whose weight equals the letter's frequency. We repeatedly choose the smallest tree roots (by weight) and merge them. The new root's weight is the sum of the two children's weights. If there are n letters in the alphabet, there are $n - 1$ merges.

```
1 Q ← C (Q is a priority queue)
2 for i = 1 to n − 1 do
3 |   z ← allocateNode()
4 |   x ← left[z] ← DeleteMin(Q)
5 |   y ← right[z] ← DeleteMin(Q)
6 |   f(z) ← f[x] + f[y]
7 |   Insert(Q, z)
8 return FindMin(Q)
```

1.3.10. Example

Initial	f:5	e:9	c:12	b:13	d:16	a:45
Merge/Reorder	c:12	b:13	f+e:14	d:16	a:45	
Next	f+e:14	d:16	c+b:25	a:45		
Next	c+b:25	(f+e)+d:30	a:45			
Next	a:45	(c+b)+((f+e)+d):55				



1.3.11. Runtime Analysis

Note that this is $O(n \log n)$ because we initially sort and then do n heap operations.

1.3.12. Remark

In an inductive proof, which character should we drop for induction? As a thought experiment, suppose we drop x_n. Then by induction, we have an optimal solution for the problem on $\{x_1, ..., x_{n-1}\}$. This is a tree with $n-1$ leaves. Where should we attach x_n? Note attaching x_n to a leaf node will create a node with one child, which cannot be optimal.

We have a better idea. Take the two lowest frequency characters x_{n-1} and x_n, and combine them into a new single character z with frequency $f(z) = f(x_{n-1}) + f(x_n)$. With x_{n-1}, x_n removed and replaced with z, we have a set of size $|C'| = n-1$. By induction, we find the optimal code tree of C'. This tree has z at some leaf. To obtain the tree for C, we attach nodes x_{n-1} and x_n as children of z. We will show that given an optimal tree for C', this new tree is optimal for C.

1.3.13. *Lemma*

> Suppose x and y are two letters of lowest frequency. Then, there exists an optimal prefix code in which codewords for x and y have the same and maximum length in that the differ in only one character.

Proof: Idea: Suppose the optimal tree T does not satisfy the claim. We will modify it by making x and y sibling leaves of maximum depth, and show that the change does not increase the total cost of the tree, which will prove the lemma.

Suppose a and b are two characters that are sibling leaves of max depth in T.

Without loss of generality, assume that

$$f(a) \leq f(b) \text{ and } f(x) \leq f(y).$$

Note because $f(x)$ and $f(y)$ are the two lowest frequencies, we can conclude that

$$f(x) \leq f(a) \text{ and } f(y) \leq f(b).$$

(x, y, a, b need not be distinct.)

First transform T into T' by swapping the positions of x and a. Since $d_T(a) \geq d_T(x)$ and $f(a) \geq f(x)$, swapping does not increase frequency times depth cost.

Then

$$\begin{aligned} B(T) - B(T') &= \sum_p [f(p)d_T(p)] - \sum_p [f(p)d'_T(p)] \\ &= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d'_T(x) + f(a)d'_T(a)] \\ &= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_T(a) + f(a)d_T(x)] \\ &= [f(a) - f(x)] \cdot [d_T(a) - d_T(x)] \\ &\geq 0 \end{aligned}$$

Therefore, if T is optimal, so is T'. Next, modify T' to T'' by exchanging y and b, and the same argument shows $B(T'') \leq B(T') \leq B(T)$. Thus T'' is an optimal tree but has x and y as sibling leaves at the maximum depth.

□

1.3.14. ***Proposition***

> Huffman's Algorithm is optimal.

Proof: We proceed by induction on the size of the alphabet $|C|$. The base case of $|C| = 2$ is trivial: we have a depth 1 tree with two leaves, each with code length 1.

By the structure lemma, the greedy choice is correct. Given input C of size n, we remove the smallest keys x, y and instead add a new key z with $f(z) = f(x) + f(y)$, getting a $(n-1)$ size set $C' = C \setminus \{x, y\} \cup z$. By induction, Huffman's Algorithm finds an optimal tree T' for C', one of whose leaves is z. We make x and y children of z, obtaining a tree T for the original input C.

□

1.4. Minimum Lateness

Lecture 4 Jan 16

1.4.1. Computational Problem

- We have n jobs to schedule on a single processor
- Each job j comes with two requirements: processing time t_j and a due date d_j
- That is, jobs do not have fixed start and finish times, and it is our scheduling algorithm's task to select their start times
- If job j were to start at time s_j, then it will finish at time $f_j = s_j + t_j$
- The *lateness* of a job is defined by

$$\ell_j = \max\{0, f_j - d_j\}$$

- The *maximum lateness* of a schedule S is

$$L(S) = \max_j \ell_j$$

- Goal: find a schedule with smallest maximum lateness $L(S)$

1.4.2. Example

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

If we do $d_3 \to d_2 \to d_6 \to d_1 \to d_5 \to d_4$, the first late job is job 1 and job 4 is also late. $n!$ possible ways to schedule jobs.

1.4.3. Example

1. Consider a strategy where we take the shortest processing time first. But if we have the tuples $(1, 100)$ and $(10, 10)$. The first job finishes at time 1 and the second job finishes at time 11, so the delay is 1. But if we did the second job first and the first job after, we have 0 delay.
2. Consider sorting by the smallest slack $d_j - t_j$ on $(1, 2)$ and $(10, 10)$. Note the first has slack 1 and the second slack 0, but we should've done the first one first.

1.4.4. Algorithm

We schedule jobs in ascending order of due dates.

```
Sort n jobs by deadline d_1 ≤ d_2 ≤ ··· ≤ d_n
t = 0
for j = 1 to n
    Assign job j to interval [t, t + t_j]
    s_j ← t, f_j ← t + t_j
    t ≤ t + t_j
output intervals [s_j, f_j]
```

1.4.5. *Proposition*

The EDD algorithm is optimal.

Proof: By contradiction, suppose there is a better algorithm S^*. Note that the greedy algorithm has $d_i \leq d_j$ if $i < j$. Thus there must be some consecutive pair in S^* such that $d_i > d_j$ if $i < j$ (an *inversion*). This reduces to the following claim: swapping inverted jobs reduces the number of inversions and does not increase max lateness.

Let ℓ be the lateness before the swap, and let ℓ' be it afterwards. Assume $d_j > d_i$. Note $\ell'_k = \ell_k \forall k \neq i, j$ and $\ell'_i \leq \ell_i$ Also

$$\begin{aligned} \ell'_i &= f'_j - d_j \\ &= f_i - d_j \\ &\leq f_i - d_i \\ &\leq \ell_i \end{aligned}$$

so S^* is not optimal.

□

1.5. Shortest Path and Minimum Spanning Tree

1.5.1. Computational Problem

Suppose we have a directed graph $G = (V, E)$ where every edge $e \in E$ has a nonnegative cost $\text{cost}(e)$. The length of a path P between two nodes u and v is the sum of the edge costs in P.

1.5.2. Algorithm

Initiailize $S = \emptyset, d(s) = 0$ and $d(u) = \infty \forall u \neq s$
Insert vertices into a minheap Q with distance label $d(u)$
while $S \neq V$ do
    $u :=$ deleteMin(Q)
    Add u to S
    for each out neighbor $v \notin S$ of u do
        $d(v) = \min\{d(u) + \text{cost}(u, v), d(v)\}$

1.5.3. *Proposition*

Djikstra's is correct.

Proof:

1. At any time $d(v)$ is the shortest path distance to $v \forall v \in S$.
2. Consider the instsant when v is added to S. Let (u, v) be the edge, with $u \in S$, that last updated $d(v)$.
3. Suppose for the sake of contradiction that $d(u) + \text{cost}(u, v)$ is not the shortest distance to v. Instead a different shorter path called P exists to v.
4. Since the path starts at S, it has to leave S at some node x. Let $y \notin S$ be the edge that goes from S to $\overline{S}$
5. But note that $d(u) + \text{cost}(u, v) \leq d(x) + \text{cost}(x, y)$
6. Since $\text{length}(y, v) > 0$, this contradicts our hypothesis that P is shorter than $d(u) + \text{cost}(u, v)$, so $d(v)$ is correct.

□

1.5.4. Computational Problem

SUppose we haev a graph $G = (V, E)$ where every edge has a cost $c(e)$. A spanning subgraph is connected and includes all vertices of G. We want to find the minimum spanning tree.

1.5.5. Algorithm

Sort the edges in increasing order and assume $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$
$S = \emptyset$
For e edge do
    if e does not create a cycle do
        $S = S \cup \{e\}$

1.5.6. *Proposition*

Kruskal's is correct

Proof: Follows from cut property

□

2. Divide and Conquer

2.1. Introduction

Lecture 5 Jan 21

2.1.1. Concept

Divide and conquer problems have the following structure:
1. Break up the problem into multiple parts
2. Solve each part (sub problem) recursively
3. Combine sub problem solutions into overall solution

2.1.2. Example

In a sorting problem we want to rearrange n elements in order. It has many applications and is often solved by divide and conquer.

2.1.3. Algorithm

This is a classic divide and conquer algorithm
1. Divide the array into two halves
2. Recursively search each half
3. Merge two halves to make the whole array sorted

Note to merge two lists, we often use a temporary array to place elements into. We can do it with only constant extra space, but it becomes much more complicated.

2.1.4. Runtime Analysis

Define $T(n)$ as the number of steps to merge sort an input of size n. Then the runtime is expressed using

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

where c is some fixed constant and $T(1) = 0$. Assume n is a power of 2.

We can show that this implies $T(n) = O(n \log n)$, but we will show this later.

2.1.5. Definition

Given a list of n numbers $a_1, a_2, ..., a_n$, an inversion is a pair (i, j) where $i < j$ but $a_i > a_j$.

2.1.6. Computational Problem

An interesting problem is to count *inversions* in a list.

For example, given $(1, 3, 4, 2, 5)$ the inversions are $3 - 2$ and $4 - 2$. There are $\binom{n}{2}$ pairs (if it's in reverse order, this is the number of out of order terms). This gives a trivial $O(n^2)$ algorithm, but can we do better?

2.1.7. Algorithm

Suppose we have an array like

`1 5 4 8 10 2 6 9 12 11 3 7`

and we split it into `1 5 4 8 10 2` and `6 9 12 11 3 7`.

Inversions in the first part:

`5-4, 5-2, 4-2, 8-2, 10-2`

and inversions in the second part:

`6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7.`

Notice if we sort those two lists, we can create two pointers traversing each list exactly once. We look at the first element at the right and move on the left until we find an element larger than the element on the list - then the number of spaces we moved is the number of inversions. This is $O(n)$ time after the merge sort (which is $O(n \log n)$) so the algorithm is $O(n \log n)$ overall.

2.1.8. Computational Problem

Suppose we want to multiply two long n digit numbers (such that the operation is not constant time in the processor).

Note the standard method is $O(n^2)$.

2.1.9. Algorithm

Suppose we have n bit numbers X and Y. Let a be the $n/2$ bit number representing the leading bits of X, and b represent the trailing bits. Note that $X = 2^{n/2}a + b$. Similarly, let c and d be the numbers corresponding to leading and trailing $n/2$ bits of Y, so $Y = 2^{n/2}c + d$.

Then

$$\begin{aligned} XY &= \left(a2^{n/2} + b\right)\left(c2^{n/2} + d\right) \\ &= ac2^n + (ad + bc)2^{n/2} + bd \end{aligned}$$

2.1.10. Example

Let $X = 4729$ and $Y = 1326$. Then $a = 47$, $b = 29$, $c = 13$ and $d = 26$. Then $ac = 611$, $ad = 1222$, $bc = 377$ and $bd = 754$. Therefore

$$XY = 611 \cdot 10^4 + 1599 \cdot 10^2 + 754.$$

2.1.11. Runtime Analysis

We split each number in 4 and we have some shifting operations, so the total number of operations is

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n).$$

This is $T(n) = O(n^2)$.

2.1.12. Algorithm

Karatsuba made an improvement to the previous algorithm. He noticed we only need ac, $ad + bc$ and bd.

In particular,

$$(a - b)(c - d) = (ac + bd) - (ad + bc).$$

Suppose we find ac, bd and $(a - b)(c - d)$ – then we get $ad + bc$. Now we've reduced the problem to 3 subproblems. This gives

$$T(n) = 3T(n/2) + O(n)$$

which gives $T(n) = O(n^{1.59})$. (Note $1.59 = \log_2 3$).

2.2. Solving Recurrence Relations

Lecture 6 Jan 23

2.2.1. Concept

One way to solve recurrence problems is to try to find a pattern.

2.2.2. Example

Suppose $T(n) = 2T(n/2) + cn$. Thus

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= 2(2T(n/2^2) + cn/2) + cn \\ &= 2^2T(n/2^2) + 2cn \\ &= 2^2(2T(n/2^3) + cn/2^2) + 2cn \\ &= 2^3T(n/2^3) + 3cn \\ &\vdots \\ T(n) &= 2^iT(n/2^i) + icn \end{aligned}$$

Now set $i = \log_2 n$ and use $T(1) = 1$. Then we see $T(n) = n + cn\log(n)$

2.2.3. Example

$$\begin{aligned}
T(n) &= 4T(n/2) + cn \\
&= 4(4T(n/2^2) + cn/2) + cn \\
&= 4^2T(n/2^2) + 2cn + cn \\
&= 4^2(4T(n/2^3) + cn/2^2) + 2cn + cn \\
&= 4^3T(n/2^3) + 2^2cn + 2cn + cn \\
&\vdots \\
&= 4^iT(n/2^i) + cn(2^{i-1} + 2^{i-2} + \cdots + 2 + 1) \\
&= 4^iT(n/2^i) + 2^icn
\end{aligned}$$

Now set $2^i = n \Leftrightarrow i = \log_2 n$ and we get $T(n) = n^2 + cn^2 = O(n^2)$.

2.2.4. Example

$$\begin{aligned}
T(n) &= 2T(n/4) + \sqrt{n} \\
&= 2\left(2T(n/4^2) + \sqrt{n/4}\right) + \sqrt{n} \\
&= 2^2T(n/4^2) + 2\sqrt{n} \\
&= 2^3T(n/4^3) + 3\sqrt{n} \\
&\vdots \\
&= 2^iT(n/4^i) + i\sqrt{n}
\end{aligned}$$

Thus we want $n = 4^i$ so $i = \frac{1}{2}\log_2 n$. So this is $\sqrt{n} + \frac{1}{2}\log_2 n \cdot \sqrt{n} = O(\sqrt{n}\log n)$.

2.2.5. Concept

Visualize the recursion as an infinite tree and figure out how to collapse it.

2.2.6. Example

Supopse $T(n) = 4T(n/2) + cn$ and $T(1) = 1$. At the ith level there are 4^n nodes but we only pay $\frac{cn}{2^i}$ for each one, so the additional work we do at each step is $\frac{4^icn}{2^i} = 2^icn$.

Then if we set $i = \log_2 n$ we get that this is $O(n^2)$.

2.2.7. Example

Our recurrence relation is $T(n) = 2T(n/2) + cn$. Our base case is $T(1) = 1$. Our inductive hypothesis is $T(n) = cn\log n$. We verify as follows:

$$\begin{aligned}
T(2n) &= 2T(2n/2) + 2cn \\
&= 2cn\log n + 2cn \\
&= 2cn\log(2n/2) + 2cn \\
&= 2cn\log(2n) - 2cn + 2cn \\
&= 2cn\log(2n)
\end{aligned}$$

2.2.8. Concept

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Suppose we have a recurrence relation of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ with } T(1) = 1$$

We are turning a problem n into a subproblems of size $\frac{n}{b}$ each. Thus at each level there are
1. $f(n)$
2. $af(n/b)$
3. $a^2 f(n/b^2)$

And in general $a^i f(n/b^i)$ problems.

The number of leaves is then $a^{\log_b n} = n^{\log_b a}$. By the recursion step we get

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{i=0}^{\log_b(n-1)} a^i f\left(\frac{n}{b^i}\right)$$

This general sum is hard to bound so we limit ourselves to the situation where f looks like

$$f(n) = \Theta\left(n^p \log^k n\right) \text{ with } p, k \geq 0 \text{ and } a \geq 1 \text{ and } b > 1 \text{ are constants}$$

This breaks down to 3 cases:

1. $p < \log_b a$. Then $n^{\log_b a}$ grows faster than $f(n)$ so
$$T(n) = \Theta\left(n^{\log_b a}\right)$$
2. $p = \log_b a$. Both terms have the same growth rates so
$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$
3. $p > \log_b a$. Then $n^{\log_b a}$ is slower than $f(n)$ so
$$T(n) = \Theta(f(n))$$

2.2.9. Example

Merge Sort has $T(n) = 2T(n/2) + \Theta(n)$.
1. $a = b = 2, p = 1$ and $k = 0$
2. So $\log_b a = 1$ and thus $p = \log_b a$
3. Case 2 applies so
$$T(n) = \Theta(n \log n)$$

2.2.10. Example

Suppose $T(n) = T(n/2) + \Theta(1)$.
1. $a = 1, b = 2, p = 0, k = 0$.
2. So $\log_b a = 0 = p$
3. Case 2 so $T(n) = \Theta(\log n)$

2.2.11. Example

1. $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \log n) \implies T(n) = \Theta(n \log^2 n)$
2. $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \implies$ Case 1 : $T(n) = \Theta\left(n^{\log 7}\right)$
3. $T(n) = 4T\left(\frac{n}{2}\right) + \Theta\left(n^{\frac{5}{2}}\right) \implies$ Case 3: $T(n) = \Theta\left(n^{\frac{5}{2}}\right)$
4. $T(n) = 2T\left(\frac{n}{2}\right) + \Theta\left(\frac{n}{\log n}\right) \implies$ No cases apply since $k = -1$

2.2.12. Example

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n$ since a non constant
- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$ since $f(n)$ negative
- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n))$ since $f(n)$ has wrong form

2.3. Matrix Multiplication

Lecture 7 Jan 28

2.3.1. Computational Problem

We want to multiply two $n \times n$ matrices. One simple divide and conquer algorithm is to divide the matrices into 4 submatrices and multiply them, i.e. if we have

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}; \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}; \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

we can calculate C via

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$

et cetera, which gives

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

which is $O(n^3)$. Is there something better?

2.3.2. Algorithm

Only 7 subproblems are needed:

$$
\begin{aligned}
P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_2 &= (a_{21} + a_{22})b_{11} \\
P_3 &= a_{11}(b_{12} - b_{22}) \\
P_4 &= a_{22(b_{21} - b_{11})} \\
P_5 &= (a_{11} + a_{12})b_{22} \\
P_6 &= a(a_{21} - a_{11})(b_{11} + b_{12}) \\
P_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}
$$

Then we can recover C as follows:

$$
\begin{aligned}
c_{11} &= P_1 + P_4 - P_5 + P_7 \\
c_{12} &= P_3 + P_5 \\
c_{21} &= P_2 + P_4 \\
c_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}
$$

2.3.3. Runtime Analysis

The recurrence looks like

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

which solves to

$$\mathbb{T}(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$$

2.3.4. Notation

Computer scientists use ω for the current best possible exponent for matrix multiplication.

2.4. Quick Sort

2.4.1. Algorithm

When we perform a quick sort we *partition* first. This means that we select an element x as a pivot and place all elements smaller than x on its left and all elements larger on its right.

The partition is easy to implement in $O(n)$ via

- Keep two pointers i and j such that
  - items to the left of i are less than x
  - items from $i+1$ to j are bigger than x
  - items to the right of j are not yet scanned

When j reaches end of array (pivot), swap with item at $i+1$

We pick a random index as our pivot. So our algorithm is

1 if $p \geq q$ then return
2 $i \leftarrow \text{random}(p, q)$
3 $r \leftarrow \text{Partition}(A, p, q, i)$
4 $\text{Quicksort}(A, p, r-1)$
5 $\text{Quicksort}(A, r+1, q)$

2.4.2. Runtime Analysis

Our recurrence is

$$T(n) = T(n_1) + T(n_2) + O(n)$$

with $n_1 + n_2 = n$. A lucky case is that

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

and an unlucky one is that

$$T(n) = T(n-1) + O(n) = O(n^2)$$

Note that fixing any position of our pivot, like always choosing the first element, fails to adversarial lists that can always give us $O(n^2)$ time.

Let $T(n)$ denote the expected runtime of QuickSort. Assume elements are distinct. If the pivot is the ith smallest element, we get $i-1$ elements in L and $n-i$ in R—call this an i split.

Thus

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{i=1}^{n}(\text{runtime with } i \text{ split}) + n + 1 \\ &= \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)) + n + 1 \\ &= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1 \\ &= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n + 1 \end{aligned}$$

Thus

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

so

$$nT(n) = (n+1)T(n-1) + 2n.$$

Thus

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\ &= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{T(2)}{3} + \sum_{i=3}^{n}\frac{2}{i} \\ &= \Theta(1) + 2\log n. \end{aligned}$$

2.5. Median and Closest Pair

Lecture 8 Jan 300

2.5.1. Definition

Given a set of n items, we define the *rank* of an item x as its position in the sorted order of the items.

2.5.2. Computational Problem

Given a set of n items and integer $1 \leq k \leq n$, find the item in the set with rank k. Assume all values are distinct. Note that if we sort them, this is trivial, but this is $O(n \log n)$—can we do better?

2.5.3. Algorithm

Note that it's easy to find items or rank 1 or n in $O(n)$ time:

1 if $|A| = 1$, return min $=$ max $= A[0]$
2 Divide A into two equal subsets A_1, A_2
3 $(\min_1, \max_1) :=$ MIN-MAX(A_1)
4 $(\min_2, \max_2) :=$ MIN-MAX(A_2)
5 if $\min_1 \leq \min_2$ then
6 | return min $= \min_1$
7 else
8 | return min $= \min_2$
9 if $\max_1 \geq \max_2$ then
10 | return max $= \max_1$
11 else
12 | return max $= \max_2$

2.5.4. Remark

If we extend this algorithm to find the rank k item it takes $O(kn)$ time, which in particular for the median takes $O(n^2)$ time. We want something with a smaller runtime that we can use to find the ideal pivot for QuickSort, for example.

2.5.5. Algorithm

1 Divide the items into $\lceil \frac{n}{5} \rceil$ groups of 5 (or less for final group) items each
2 Find the median of each group (brute force)
3 Use SELECT to recursively find the median of the $\frac{n}{5}$ group medians
4 Partition the input by using median of median as pivot
5 Suppose low side of partition has s elements and high side has $n - s$ elements
6 If $k \leq s$, recursively call SELECT(k) on low side; otherwise, recursively call SELECT($k - s$) on high side

2.5.6. Runtime Analysis

There are $\frac{n}{5}$ groups and in the partition step, half of them have medians less than x. The group medians and elements smaller form the top left quadrant and each have $\frac{3n}{10}$ elements. Symmetrically, the bottom right consists of $\frac{3n}{10}$. At most, we have $\frac{7n}{10}$ unknown.

Thus our recursion looks like

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

So

$$\begin{aligned} T(n) &\leq c\frac{n}{5} + 7c\frac{n}{10} + bn \\ &\leq 9c\frac{n}{10} + bn \end{aligned}$$

If we choose $c = 10b$ note that $cn \geq 9c\frac{n}{10} + bn$, so T has running time $O(n)$.

2.5.7. Remark

Why not divide into groups of three instead? Then the recursion looks like

2.5.8. Computational Problem

Given n points in the plane, find a pair with the smallest Euclidean distance between them.

2.5.9. Algorithm

Draw a vertical line L so that there are roughtly $\frac{n}{2}$ points on each side. We then find the closest pair in each side recursively.

2.5.10. Remark

The previous algorithm fails because there could be points close to the line that are closer together. Thus look at the boundary of the line and consider some distance δ, which δ is the minimum of the distances in each of the partitions. Now in our δ width window, for each side we can say that all points are at least δ distance apart. (This last claim follows from creating $\frac{\delta}{2}$ boxes within the partition).

Lecture 9 Feb 4

2.5.11. Algorithm

1 Compute separation line L such that half the points are on on side and half on the other
2 $\delta_1 = \text{ClosestPair}(\text{left half})$
3 $\delta_2 = \text{ClosestPair}(\text{right half})$
4 $\delta = \min\{\delta_1, \delta_2\}$
5 Delete all points further than δ from separation line L
6 Sort remaining points by y coordinate
7 Scan points in y order and compare distance between each point and 11 neighbors. Update δ if any distances are less than δ
8 Return δ

Note that this is $O(n \log^2 n)$.

3. Dynamic Programming

3.1. Introduction

Lecture 10 Feb 11

3.1.1. Remark

Some problems are beyond the reach of greedy algorithms and divide and conquer. For example:

1. Weighted interval scheduling — each of the n jobs has an associated weight v_i, and the goal is to maximize the total weight of the schedule.
2. Find the shortest path from s to t with at most k hops, where k is specified at input. The greedy style Djikstra's Algorithm doesn't work.

3.1.2. Concept

A technique where a problem is broken down into subproblems and the results are saved. Note the time complexities are generally worse than greedy algorithms.

3.1.3. Computational Problem

- We have a single processor and a list of n jobs
- Every job has a start time s_j, finish time f_j and weight w_j
- A feasible schedule is a subset of non overlapping jobs
- We want to find a feasible schedule with *maximum possible total weight*

3.1.4. Concept

Suppose we list the jobs in increasing finish time order. Define $\mathrm{OPT}(j)$ as the optimal solution to the subproblem consisting of jobs $1, 2, \ldots, j$. This gives us n subproblems of increasing sizes; $\mathrm{OPT}(1)$ is trivial to compute, and $\mathrm{OPT}(n)$ is the final answer we want. Note we want to compute $\mathrm{OPT}(j)$ using $\mathrm{OPT}(i)$ for $1 \leq i \leq j-1$.

Now to compute $\mathrm{OPT}(j)$, we need to consider job j. Note that we have a binary choice: we either accept job j or we don't. If job j is included, then $\mathrm{OPT}(j)$ cannot include any previous jobs overlapping with it. If we don't include it, $\mathrm{OPT}(j)$ is free to include any previous jobs.

3.1.5. Notation

Let $p(j)$ be the largest index $i < j$ such that job i does not overlap with job j.

3.1.6. Algorithm

We write $\mathrm{OPT}(j) = \max\{v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1)\}$. Note the first term accounts for choosing j and the second doesn't. We claim that together with $\mathrm{OPT}(0) = 0$, this completes the DP solution.

3.1.7. Remark

Note that if we tried to write a recursive algorithm like so:

```
Sort jobs so that f_1 ≤ f_2 ≤ ··· ≤ f_n
Compute p(1), p(2), ..., p(n)
Compute OPT(j):
    if j = 0
        return 0
    else
        return max{v_j + OPT(p(j)), OPT(j − 1)}
```

This doesn't work!! The number of subproblems solved grows like the Fibonacci sequence, giving us exponential runtime, since we are not caching the results.

3.1.8. Concept

Memoization means we store the result of each subproblem in a table so that we can lookup as needed.

This gives a better algorithm:

```
Sort jobs so that f_1 ≤ ··· ≤ f_n
Compute p(1), ..., p(n)
for j = 1 to n, set M[j] = ∅
Initialize M[0] = 0.
Return M-ComputeOPT(n)
```

M-ComputeOPT(j):

```
if M[j] empty
    M[j] = max{v_j + M-ComputeOPT(p(j)), M-ComputeOPT(j − 1)}
return M[j]
```

3.1.9. Runtime Analysis

Sorting by finish time and compute p values takes $O(n \log n)$ time.

Each entry of the array is computed only once, and each call either takes $O(1)$ time or calls two subproblems.

Thus overall the algorithm is $O(n \log n)$.

3.1.10. Remark

We can also write the DP algorithm as a *bottom-up* unwound recursion, where entries are computed in increasing problem size order:

```
Sort jobs st f_1 ≤ ··· ≤ f_n
Compute p(1), ..., p(n)
Iterative-WIS {
    M[0] = 0
    for j = 1 to n
        M[j] = max{v_j + M[p(j)], M[j − 1]}
}
```

3.2. Matrix Chain Product

Lecture 11 Feb 13

3.2.1. Computational Problem

Suppose we have a sequence of matrices $M_1, ..., M_n$. We want to find the product $M_1 \times \cdots \times M_n$. Note these are not necessarily square matrices—we only know that adjacent matrices agree on dimension.

3.2.2. Algorithm

```
Let A be a p × q matrix and B a q × r matrix
for i = 1 to p
    for j = 1 to r
        C[i, j] = 0
    for k = 1 to q
        C[i, j] += A[i, k] · B[k, j]
return C
```

The cost of this multiplication is $p \times q \times r$.

3.2.3. Example

Consider a 10×100 matrix M_1, a 100×5 matrix M_2, and a 5×50 matrix M_3. If we do $((M_1 M_2) M_3)$ we have cost $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$. But if we do $(M_1 (M_2 M_3))$ we have cost $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$.

3.2.4. Remark

If we tried to brute force all possibilities, we get the Catalan number $\binom{2n-2}{n-1} \approx 4^n$.

3.2.5. Algorithm

We can specify the input compactly as $n+1$ numbers $p_0, ..., p_n$ where M_1 is a $p_0 \times p_1$ matrix, M_2 is a $p_1 \times p_2$ matrix, and so on. In general, M_i is a $p_{i-1} \times p_i$ matrix because all neighboring matrices agree on the dimension.

3.3. Optimal Binary Search Trees

Lecture 12 Feb 18

3.3.1. Computational Problem

Our input is a list of n keys (words) $w_1, ..., w_n$ along with their respective relative search frequencies $p_1, ..., p_n$ where $0 \leq p_i \leq 1$ and $\sum_{i=1}^{n} p_i$. We want to minimize the (expected) total access cost. Accessing a key at depth d has search cost $d+1$, so if the word w_i is placed at depth d_i in the tree, the total search cost is

$$\sum_{i=1}^{n} p_i \cdot (d_i + 1).$$

3.3.2. Algorithm

Define subproblems of the form $[i, j]$ for all $1 \leq i \leq j \leq n$. Let $S(i, j)$ be the optimal search tree cost for the subproblem $(w_i, ..., w_j)$. We now need a recurrence to compute $S(i, j)$ using smaller subproblems. The tree for $S(i, j)$ must use some word in $(w_i, ..., w_j)$. Suppose that word is w_r for $i \leq r \leq j$.

We then have the following recurrence: $S(i, j) = p_r + S(i, r-1) + S(r+1, j) + \sum_{k=i}^{r-1} p_k + \sum_{k=r+1}^{j} p_k$.

- The root w_r has depth 0 and search cost 1, so it contributes $p_r \cdot 1$ to the overall cost.
- $S(i, r-1)$ and $S(r+1, j)$ are optimal solutions for their subproblems, but now their subtrees *have become children of* w_r, and so depth of every node has increased by 1, which accounts for the last term.
- The sum simplifies to

$$S(i, j) = S(i, r-1) + S(r+1, j) + \sum_{k=i}^{j} p_k$$

- Note the last term does not depend on the two subtrees, so to minimize $S(i, j)$ we must also use optimal solutions $S(i, r-1)$ and $S(r+1, j)$
- Finally, as before, we do not know r, but it must be one of w_i and w_j, so take the minimum over those possibilities:

$$S(i, j) = \min_{i \leq r \leq j} \left\{ S(i, r-1) + S(r+1, j) + \sum_{k=i}^{j} p_k \right\}.$$

- We have an n^2 size table and each entry takes $O(n)$ time, so the total runtime is $O(n^3)$.

3.4. k Hop Shortest Path

3.4.1. Computational Problem

Suppose we have a directed graph $G = (V, E)$, where each edge (u, v) has a positive weight (length) $\ell(u, v)$. Given a source node s and integer $k \geq 1$, we want to find the *shortest path from s to each v using at most k hops.*

3.4.2. Algorithm

Our choice of subproblems is the following: for all nodes $v \in V$ and $j \leq k$:

$$d(v, j) = \text{shortest path from } \ s \text{ to } v \text{ using at most } j \text{ hops.}$$

Initially define $d(v, 0) = \infty$ for all $v \neq s$. The recurrence for computing $d()$ values is

$$d(v, j) = \min\left\{d(v, j-1), \min_{(u,v)\in E}(d(u, j-1) + \ell(u, v))\right\}.$$

We arrive at v from some previous node u along a shortest $(j-1)$ hop path, and add one more hop and length $\ell(u, v)$; or there is no better j hop path than a $j-1$ hop path.

3.4.3. Runtime Analysis

We fill entries of an $n \times k$ size array d column by column. The first column stores shortest paths using at most 1 hop. The jth column is computed using only entries from the $j-1$ column. The kth column of this array contains shortest k hop distances from s to every v. Since each array entry takes time $\propto \deg(v)$, it takes at must $O\left(\sum_v \deg(v)\right) = O(E)$ time to compute them. So, the algorithm runs in time $O(|V| \cdot |E|)$.

3.4.4. Remark

If we instead wanted the longest path, we might try to replace the mins with maxes. But this problem is significantly harder, since if there are cycles, the DP won't recognize them but they clearly make longest paths infinite.

3.4.5. Algorithm

Although this doesn't work in the general case due to cycles, if the input graph G is *acyclic*, then the DP corectly finds the longest paths. We can relabel the vertices of G as $v_1, ..., v_n$ so that for every edge $(v_i, v_j) \in E$, we have $i < j$. Such an ordering can be found by Topological Sort. Now we process vertices in this order and compute longest paths using

$$d(v_j) = \max_{(v_i,v_j)\in E}\{d(v_i) + \ell(v_i, v_j)\}$$

Note since we visit each edge exactly once, our time complexity is $O(|E|)$.

3.5. All Pairs Shortest Path Problem (APSP)

3.5.1. Computational Problem

Given a graph $G = (V, E)$ with edge weights $w(e)$, we want to compute the shortest path distances between all pairs of vertices. Our number of vertices is $n = |V|$ and number of edges is $m = |E|$.

3.5.2. Remark

A naive algorithm runs $O(n^2)$ separate shortest path algorithms, one for each pair. If all edge weights $w(e)$ are positive, then we can use Djikstra; otherwise use Bellman Ford.

Then the total runtime is $O(nm \log n)$ or $O(n^2 m)$ depending on which one we use.

3.5.3. Algorithm

List vertices in any order, labeled $1, \ldots, n$. Define $d(i, j, k)$ by the length of the shortest path in G from i to j whose intermediate nodes are all from the subset $\{1, 2, \ldots, k\}$.

We initialize $d(i, j, 0) = w(i, j)$; if there is no edge from i to j, assign $d(i, j, 0) = \infty$. The final distances we want are $d(i, j, n)$.

To compute $d(i, j, k)$, consider the node k.

- If the shortest path does not go through k, we have $d(i, j, k) = d(i, j, k-1)$.
- If it does go through k, the shortest path through k is the shortest path from i to k together with the shortest path from k to j. Thus we have $d(i, j, k) = d(i, k, k-1) + d(k, j, k-1)$.

```
1 D = W
2 for k = 1 to n
3 |  for i = 1 to n
4 |  |  for j = 1 to n
5 |  |  |  d(i, j, k) = min{d(i, j, k−1), d(i, k, k−1) + d(k, j, k−1)}
6 return D
```

4. Complexity Theory

4.1. P vs NP

Lecture 13 Feb idk

4.1.1. Remark

For some problems, no polynomial time algorithm will ever be found. The collection of such problems is very large. These problems are called **NP-Complete**, and a polynomial time solution for *any* problem in this class will imply polynomial time algorithms for all NP-Complete problems.

Informally, **NP** problems are problems whose solutions have a solution verifiable in polynomial time.

4.1.2. Example

For example,
- does a graph G have a simple (loopless) path of length K?
- is a number composite or prime?
- Does a graph have a vertex cover of size C?

Non-examples
- Graph G does not contain a simple path of length more than K
- Does White have a winning strategy in chess?

4.1.3. Remark

Clearly, $P \subseteq NP$, where P is the set of polynomial algorithms. In a sense, NP-Complete problems can be thought of as the "hardest" prolems in NP, since a solution of any of one f them in polynomial time implies they can all be solved in polynomial time. So either $P = NP$ or $P \neq NP$.

4.1.4. Concept

To prove a problem is NP-Complete, the main tool is **reduction**.

Reduction from a problem A to a problem B is a polynomial time algorithm R that transforms inputs of A to equivalent inputs to B. When such a reduction exists, we write $A \leq B$. More precisely, given $x \in A$, the algorithm R produces an input $R(x) \in B$ such that x is TRUE for A if and only if $R(x)$ is TRUE for B.

4.1.5. Remark

Given $A \leq B$:
- If B is known to be easy, A is easy too
- If A is known to be hard, B is hard too.

For NP-Completeness reductions, we use the second implication.

4.1.6. Example

In each case, the first problem is not NP Complete and the second variation is.

Trees

- MST: given a weighted graph and integer K, is there a tree of weight $\leq K$ connecting all the nodes?
- Traveling Salesman Problem: given a weighted graph and integer K, is there a (simple) cycle of weight $\leq K$ visiting all the nodes?

Tours

- Euler Tour: given a directed graph, is there a closed path visiting every edge exactly once?
- Hamilton Tour: given a directed graph, is there a closed path visiting every node exactly once?

Circuits

- Given a boolean circuit and 0/1 values for inputs, is the output TRUE?
- Given a boolean circuit, is there a 0/1 setting of inputs for which the ouput is 1?

Missed Lecture Feb 25

4.2. Examples of NP-Complete Problems

Lecture N Feb 27

4.2.1. Computational Problem

Given a directed graph $G = (V, E)$, is there a simple cycle T visiting each vertex of V exactly once?

We will reduce 3SAT to this problem. Recall 3SAT looks like $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i = \left(x_{i_1} \vee \overline{x}_{i_2} \vee x_{i_3}\right)$.

We begin by describing a graph with 2^n different cycles, each one corresponding to one of the 2^n possible truth assignments for the x_i's. Now construct n paths $P_1, ..., P_n$, where P_i consists of nodes $v_{i,1}, ..., v_{i,b}$ for $b = 3k + 3$, where we recall that k is the number of clauses.

Note that a Hamiltonian cycle in this graph looks like taking P_i left-right or right-left, then moving to P_{i+1}. Thus there are 2 choices for each P_i and there are n of them, so there are 2^n possible Hamiltonian cycles.

Now each P_i represents whether we traveled left or right. I.e. $x_1 \vee \overline{x_2} \vee x_3$ means we traveled right along P_1, left along x_2, and right along x_3. I.e. for each clause c_j, reserve two adjacent node positions $3j$ and $3j + 1$ in each path where c_j can be spliced.

1. If x_i is not negated, then add edges $v_{i,3j} \to c_j$ and $c_j \to v_{i,3j+1}$
2. If x_i is negated in c_j, then add edges $v_{i,3j+1} \to c_j$ and $c_j \to v_{i,3j}$.

This completes the construction. Note the number of nodes is $(3k + 3)n + 2 + k$.

In the forward direction, suppose 3SAT is satisfiable. Then we form a Hamiltonian cycle following our plan. Since each clause c_j is satisfied, there will be at least oen path P_i going in the correct direction relative to c_j, so we can correctly splice c_j into its edges.

Conversely, suppose the graph has a Hamiltonian cycle. If the cycle enters a node c_j on an edge from $v_{i,3j}$, it must depart on an edge $v_{i,3j+1}$. If not, then $v_{i,3j+1}$ will have only one unvisited neighbor left, namely $v_{i,3j+2}$, and so the tour will not be able to visit this node and maintain the Hamiltonian property. Symmetrically, if the tour enters $v_{i,3j+1}$, it must depart immediately to $v_{i,3j}$. Thus, direction of travel along each path P_i tells us how to set x_i to satisfy the formula.

4.2.2. Computational Problem

Given a set of integers $a_1, ..., a_n$, and an integer parameter k, decide if there is a subset of integers that sum exactly to k. It's easily in NP: just check if the numbers in the solution sum to k. We will reduce the vertex covering problem to this problem. The outline is as follows:

1. Start with G and parameter k
2. Create a sequence of integers and parameter k'
3. Prove that G has a vertex cover of size k if and only if some subset of integers sum to k'

Let C be the set of vertices corresponding to x_i's in the sum and $E' \subset E$ be the

4.2.3. Computational Problem

Given a set of integers $a_1, ..., a_n$, determine if there is a partition into two subsets with equal sums. Is there a subset $I \subseteq \{1, 2, ..., n\}$ such that $\sum_{i \in I} a_i = \sum_{i \in I} a_i$. Observe this is just a special case of subset sum where k is half the total sum, so it's NP-Complete.

4.2.4. Computational Problem

Given items of size $a_1, a_2, ..., a_n$ and an unlimited supply of bins, each of size B, we want to pack items into the fewest possible bins. The decision version is to decide if the items can be packed into k or fewer bins. The problem in NP and for NP-Completeness we reduce the Partition problem to it. Given an instance of Partition, create items of size $a_1, ..., a_n$. Then if we have $k = 2$ bins with each one having capacity $\frac{S}{2}$ where $S = \sum_i a_i$, we can see the reduction.

4.2.5. Computational Problem

Subset sum is a special case of the Knapsack Problem. Given a set of n items, each with an integer size s_i and an integer value v_i, and a knapsack of size K, the goal is to select a subset of items with maximum total value whose size is $\leq K$. Given an instance of subset sum $\{a_1, ..., a_n, k\}$, set each size and value equal to a_i and set the knapsack size to be k.

4.2.6. Remark

But we saw a dynamic programming algorithm to solve the Knapsack Problem, so how is this possible? This algorithm runs in $O(nk)$, but we should actually be considering the size of the input length of k, which is $\log k$. Thus the true running time is like $O\left(n2^k\right)$. Still, this makes the problem "weakly NP Complete" (and same for the subset sum and partition).

5. Approximation Algorithms

5.1. ρ-Approximation Algorithms

Lecture n Mar 4

5.1.1. Computational Problem

Given an NP-Complete problem, we want a way that finds a good solution in polynomial time (note finding the optimal is impossible in polynomial time). Thus, we can allow the output to be suboptimal, but want a guarantee on the solution quality. To do this, we can use the ratio between the true optimum and the algorithm's worst case solution.

5.1.2. Concept

Our goal is to minimize a function. Denote our approximation algorithm by A and a problem instance by x. We will use cost(A, x) for the quality of A's solution for x, and cost(OPT, x) for the optimal solution. Thus our measure of A's approximation quality is

$$\frac{\text{cost}(A, x)}{\text{cost}(\text{OPT}, x)}$$

The smallest value is 1, in which $A = \text{OPT}$.

Note that this varies over x, but we want to consider worst case scenarios. Thus for a particular input size n, we want to consider

$$\rho(n) = \max_x \frac{\text{cost}(A, x)}{\text{cost}(\text{OPT}, x)}.$$

5.1.3. Example

Recall the vertex cover problem. We attempt to find an approximate solution.

Note cost(A) is the size of the vertex cover found by A, and cost(OPT) is the size of the vertex cover found by OPT. Then $\rho(n)$ is the worst case ratio of $\frac{\text{cost}(A)}{\text{cost}(\text{OPT})}$ over all n node graphs.

One clear problem is that it's not clear how we can even know the denominator. We can try to estimate it.

Two natural greedy algorithms are:

5.1.4. Algorithm

```
while graph nonempty:
    choose an edge (u, v)
    add both u and v to cover C
    delete all edges covered by u and v
return C
```

5.1.5. Algorithm

```
while graph nonempty:
    choose v of maximum degree
    add v to cover C
    delete all edges covered by v
return C
```

We begin by analyzing the second algorithm. Consider a bipartite graph $G_n = (L + R, E)$, where
1. L is a set of n vertices $1, 2, ..., n$
2. For each $i = 2, 3, ..., n$, add $|R_i| = \lfloor n/i \rfloor$ vertices where each vertex of R_i is connected to i distinct vertices of L
3. Vertices of R_i have degree i; R_2 has $n/2$ vertices, each of degree 2, and so on.

Observe Algorithm 2 will pick all vertices of R, starting with R_n. THe size is $\frac{n}{2} + \frac{n}{3} + \cdots + 1 = \Omega(n \log n)$. But, the optimal can just pick all vertices of L, which gives a vertex cover of size n. Therefore, the worst case approximation ratio is at least $\rho(n) = \Omega(\log n)$ which goes to ∞ as $n \to \infty$.

You can also show an upper bound: $\rho(n) = \Theta(\log n)$, which can come from greedy Set Cover analysis.

5.1.6. *Theorem*

> Algorithm 1 has $\rho(n) = 2$.

Proof: Let A be the set of edges picked by the greedy algorithm.
1. No two edges in A share a vertex, so OPT must have size at least $|A|$.
2. The size of the greedy set cover is 2 $|A|$; it adds 2 vertices for each edge of A.
3. Therefore, we always have $\rho(n) \leq 2$.

Then it's easy to show examples with ratio 2, so $\rho(n) = 2$.

□

5.1.7. Example

We assume that edges of G obey the triangle inequality. Now take the maximum path length in the graph: we can add all missing edges to the graph and give them cost $w(x, y)$ equal to the length of the shortest path in G between them. Thus we can always make our graph complete.

We can approximate T using the MST. Pick an arbitrary node as the start point and do an in-order traversal of T. This is not a proper tour since it revisits vertices, so we modify T' so that if we reach a vertex that has already been visited, we find the shortest edge to another element in the MST.

Now observe

$$\text{cost}(\text{tour}) \leq \text{cost}(T') \leq 2\ \text{cost}(\text{MST}) \leq 2\ \text{cost}(\text{TSP})$$

Therefore we have $\rho(n) = 2$.

5.1.8. Remark

This works for the triangle inequality case, but what about when this does not apply, i.e., in a complete graph?

Lecture n Mar 6

5.1.9. *Theorem*

> Unless $P = NP$, for the TSP without the triangle ineuqality, there is no poly time algorithm with approximation ratio $\rho(N) \leq M \forall M < \infty$.

Proof: We show that if a TSP can be approximated within any fixed ratio in polynomial time, then we can solve the HAM CYCLE problem in polynomial time.

Given an instance $G = (V, E)$ of HAM, construct a TSP instance G' as follows. The vertices and edges of $G' = (V, E)$ are exactly the same as those of G. The graph G does not have edge weights, but for G' we assign edge weights as follows. Set $w(e) = 1$ if $e \in E$, and $w(e) = (nM + 1)$ if $e \notin E$. Now we wonder if G' contains a TSP of cost $\leq n$. If G contains a HAM cycle, then simply using the edges of that cycle we get a TSP of cost n in G'. The approximation algorithm A, in order to deliver its approximation guarantee, must return a tour with cost $\leq nM$. Since each edge not in G has cost $> nM$, the cycle cannot use that edge. So the only tours that lead to acceptable approximation are the HAM cycles in G.

Conversely, if TSP returns a tour that costs more than nM, it must be forced to use at least one edge of weight $(nM + 1)$, which means G does nto contain a HAM cycle.

□

5.1.10. Example

Recall the maximum clique problem is that given a graph $G = (V, E)$, we want to find the largest subset of nodes in which every pair has an edge between them.

In particular, suppose a graph contains a clique of size $n/10$. Note that in the greedy we can return two vertices joined by an edge, so it returns 2.

No one has been able to find an algorithm with an approximation ratio better than $n^{1-\varepsilon}$ for ε nonzero. Also, a theoretical result shows that if there exists a poly time algorithm that can approximate the clique within factor $O(n^{1-\varepsilon})$, then $P = NP$.

5.1.11. Computational Problem

Suppose we have items some amount of items and some weights. We want to select a subset of those items $I \subseteq \{1, 2, ..., n\}$ such that

1. $\sum_{i \in I} w_i \leq W$
2. $\sum_{i \in I} v_i$ is maximized

The problem is called the 0 / 1 Knapsack because inclusion of items is binary. Note that Subset Sum is a special case of knapsack where $w_i = v_i \forall i$, implying the Knapsack is NP-complete.

5.1.12. Algorithm

Define subproblems $A[i, w]$ for all $0 \leq i \leq n$ and $0 \leq w \leq W$. Initialize $A[0, w] = A[i, 0] = 0$ for all i and w. We then have the DP recurrence relation

$$A[i, w] = \max\{A[i-1, w], v_i + A[i-1, w-w_i]\}$$

We don't select item i in the first entry, and in the second we select it. This is done in $O(nW)$ total time; however, this is not polynomial time. It should really be a polynomial function of n and $\log W$, the size of the input. This is also why the subset sum and knapsack problems are called weakly NP complete: for small values they have polynomial algorithms.

5.1.13. Example

One greedy algorithm is to sort the items in decreasing order of value and add to the knapsack if they fit. This gives an arbitrarily large ratio, because we can set items to have weights $2, 3, ..., n$ each with weight $W - 1$, and another item 1 with weight W, whereas the optimal algorithm would choose the $n - 1$ items but we choose 1 item.

We can also try sorting by ratio v/w, but this also results in an arbitrarily large ratio, since we can choose value $1 + \varepsilon$.

However, a third approach is to run the first greedy algorithm, then run the second greedy algorithm, and take whichever has the better solution. Amazingly, this gives $\rho(n) = 2$

5.1.14. ***Proposition***

The combined greedy approach fulfills $\rho(n) = 2$.

Proof: First ignore items with $w_i > W$ since they can never fit. Next, modify our greedy algorithms so that they stop at the first item that doesn't fit in the knapsack. Let V_1, V_2 be the values returned by the greedy algorithms, respectively. Notice that if $V_1 + V_2 \geq \text{OPT}$, we would have $\max\{V_1, V_2\} \geq \frac{\text{OPT}}{2}$, the result we want to obtain.

Let I be the subset of items picked by the ratio greedy algorithm, and let j be the first item that didn't fit. Then observe

$$\sum_{i \in I} v_i = V_2$$

and

$$V_1 \geq v_j.$$

Observe this second claim follows from the fact that the first item the first greedy algorithm picks up must be at least as valuable as v_j.

Observe OPT cannot exceed $V_2 + v_j$, so V_2 or v_j must be at least OPT $/2$.

□

5.1.15. Computational Problem

Suppose we have a set of m identical machines $M_1, \ldots, M_m$ and a set of n jobs, where job j needs p_j time for processing. The goal is to schedule the jobs on these machines in a way such that we minimize the latest completion time. Formally, suppose our algorithm assigns the subset of jobs $A_i \subseteq \{1, 2, \ldots, n\}$ to machine M_i. Then, the finish time for M_i is $T_i = \sum_{j \in A_i} p_j$. We will call T_i the *load of machine* M_i. The maximum load across all machines is called the **makespan** of the algorithm, that is, $T = \max_i\{T_i\}$. This is an NP-complete problem.

5.1.16. Algorithm

List jobs in some order, then assign the next job j to the machine with the smallest current load.

```
for j = 1, ..., n
    Let M_i be the machine with the current minimum load T_i
    Add job j to machine i; i.e., set A_i = A_i ∪ {j} and T_i = T_i + p_j
```

5.1.17. Example

Suppose we have 3 machines with sizes $2, 3, 4, 6, 2, 3$. Our assignment is $A_1 = \{2, 6\}, A_2 = \{3, 2\}, A_3 = \{4, 3\}$, with makespan 8, whereas the optimal is 7, achieved by $(3, 4), (6), (2, 2, 3)$.

What is the worst case approximation of this algorithm?

Lecture N Mar 10

5.1.18. ***Proposition***

The worst case approximation is $\rho(n) = 2$.

Proof: Let T be the makespan of the greedy approach, and let T^* be the optimal makespan. One possible lower bound on T^* is

$$T^* \geq \frac{1}{m}\sum_j p_j.$$

Further, $T^* \geq \max_j\{p_j\}$.

Let M_i have the max load in greedy assignment, and j be the last job assigned to M_i. Then the load of M_i, given by $(T_i - p_j)$, was the smallest among all machines when j was assigned. Thus adding up all the loads we have

$$m(T_i - p_j) \leq \sum_{k=1}^{n} p_k \Longrightarrow T_i - p_j \leq \frac{1}{m}\sum_{k=1}^{n} p_k$$

so

$$T_i \leq p_j + \frac{1}{m}\sum_{k=1}^{n} p_k \leq 2T^*.$$

□

5.1.19. Remark

Note there is a better approximation with a different greedy algorithm: if we sort in decreasing length order, we can show $\rho(n) = \frac{3}{2}$.

5.1.20. Computational Problem

Suppose we have N items of sizes $s_1, s_2, \ldots, s_N$, where we assume $0 < s_i \leq 1$; we also have an infinite supply of *unit size* bins. There are two variations:

- online bin packing: items arrive one at a time (unordered) and each must be put in a bin before considering the next item
- offline bin packing: all items given upfront

5.1.21. ***Lemma***

There exist inputs that can force any online bin packing algorithm to use at least $\frac{4}{3}$ of the optimal number of bins.

Proof: Consider an input sequence consisting of
- I_1: a sequence of M items of size $\frac{1}{2} - \varepsilon$, possibly followed by
- I_2: a sequence of M items of size $\frac{1}{2} + \varepsilon$

Simultaneously, suppose the online algorithm has used b bins after processing I_1. The optimal clearly uses $\frac{M}{2}$ bins, so if the online algorithm beats the ratio we have

$$\frac{b}{M/2} < \frac{4}{3} \Leftrightarrow \frac{b}{M} < \frac{2}{3}.$$

After all items have been processed, note all items in I_2 have size $> \frac{1}{2}$, so every bin created after the first b bins will have exactly one item in it. Thus, the first b bins can have 2 items and the remaining bins have 1 item each, packing $2M$ requires at least $2M - b$ bins. We know that optimal uses M bins, and if it beats the $\frac{4}{3}$ ratio then $(2M - b) < \frac{4}{3}M \Leftrightarrow \frac{b}{M} > \frac{2}{3}$. Contradiction.

□

5.1.22. Algorithm

When processing the next item, check if it fits in same same bin as the last item; otherwise, start a new bin.

5.1.23. ***Theorem***

Suppose optimal uses M bins. Then Next Fit uses at most $2M$ bins, and there are inputs that force Next Fit to use $2M - 2$ bins.

Proof: The sum of items in neighboring bins is > 1 clearly. Because at most half the space is wasted, Next Fit uses at most $2M$ bins. For a lower bound, consider a sequence where $s_i = \begin{cases} \frac{1}{2} \text{ if } i \text{ odd} \\ \frac{2}{N} \text{ if } i \text{ even} \end{cases}$ (assuming $4 \mid N$). The optimal will use $\frac{N}{4} + 1$ bins, but Next Fit will use $\frac{N}{2}$ bins.

□

5.1.24. Algorithm

Check all previous bins to see if next item will fit. Start a new bin when it does not.

The upper bound is $1.7M$ bins. For a lower bound, consider the sequence of $6M$ items of size $\frac{1}{7} + \varepsilon$, followed by $6M$ items of size $\frac{1}{3} + \varepsilon$, followed by $6M$ items of size $\frac{1}{2} + \varepsilon$. We should pack each bin from one from each group, since $\frac{1}{7} + \frac{1}{3} + \frac{1}{2} + 3\varepsilon < 1$.

But first fit will give $(1 + 3 + 6)M = 10M$ bins, giving a lower bound of $\frac{10}{6}$ on the approximation ratio.

5.1.25. Algorithm

Place the next item in the tightest spot (that is, so that the smallest empty space is left.)

The same analysis for First Fit also applies to Best Fit.

5.1.26. Remark

Note that even the offline bin packing problem is NP-Complete, so we look at approximation algorithms. Consider offline analogues of first fit and best fit, called first fit decreasing and best fit decreasing.

5.1.27. *Theorem*

First Fit Decreasing uses at most $\frac{4M+1}{3}$ if the optimal number is M.