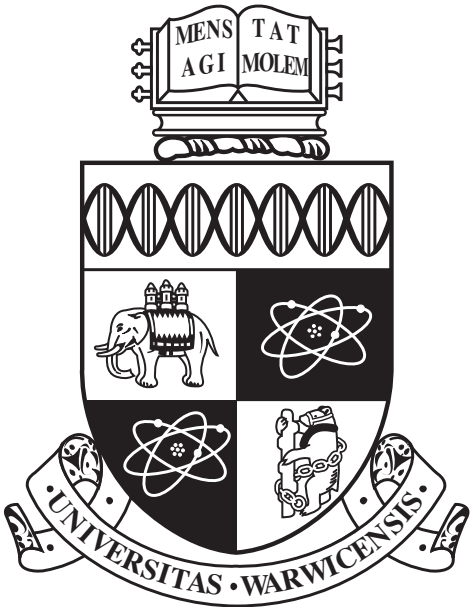
University of Warwick
Department of Computer Science

CS260

Algorithms



Cem Yilmaz
November 28, 2022

Contents

1 Revision

1.1 Past Paper

- 2019 - questions 1, 5a, 6e, 7, 8
- 2020 - questions 1a, 5d-e, 6, 7a-c, 8c-d
- 2021 - questions 1

1.2 Books

Algo design by eva jordan chapters 4,5,6,8,7, 4.1 background using chapters 2,3

2 Greedy Algorithms

Typically in a greedy algorithm:

- They are efficient to compute
- Most 'greedy rules' yield incorrect algorithms. This is why we justify correctness.

2.1 Greedy Rules

1. Earliest start time first (ESTF)
2. Shortest first (SF)
3. Fewest incompatibilities first (FIF)
4. Earliest finish time first (EFTF)

3 Interval Scheduling

The point of the interval scheduling problem is that we attempt to find disjoint intervals. That is, where an interval is a "job", we try to find which jobs are compatible, as you cannot be in two rooms at the same time. The goal is to maximise the number of jobs.

```
Initialise the set I where I = [n] =^{def} {1, 2, ..., n}. Let us initialise the answer set A = ∅.
while I ≠ ∅ do:
  pick i ∈ I using a "greedy" rule
  add i to A
  delete i from I together with all jobs that are incompatible with job i.
return A.
```

Listing 1: Interval Scheduling

- Input - n open intervals where intervals denoted by $(s(1), f(1)), (s(2), f(2)), \ldots (s(n), f(n))$ where $s(n) < f(n)$ and $f(n) = s(n+1)$.
- Output - $S \subseteq [n] =^{\text{def}} \{1, 2, \ldots, n\}$

Q: how to check ine $O(1)$ time of jobs i, j are compatible (disjoint)
$s(i) > f(j), s(j) > f(i)$

3.1 Algorithm Correctness

1. Fact 1 - A is compatible, that is, whenever we add i, we remove incompatibilities, therefore A is compatible.
2. Fact 2 - is a theorem which is found below.
3. Fact 3 -"A stays ahead of C"

Theorem 3.1. *A is optimal, that is, largest size A that is compatible.*

Proof. One way to show that is that the answer A is equal to O, the optimal set. However, this idea is not too good since O is not necessarily unique.
However, there is a work around. It is to show that $|A| \geq |C|$ for any compatible set C. To prove this, we use mathematical induction. We show that "A stays ahead of C ". Let $A = \{a_1, a_2, \ldots, a_k\} \subset [m]$. Now let us consider $C = \{c_1, c_2, \ldots, c_m\}$, which is any compatible set. The conclusion is then, for finishing time $f(a_i) \leq f(c_i)$ for all $i = 1, 2, \ldots, m$. We want to prove that m cannot exceed k. □

We now prove fact 3 by induction.

Proof. Let us consider base case $i = 1$. We note that a_1 was selected first, therefore, by the greedy rule, a_1 has earliest finish time (EFT) of all jobs from ESTF. That is, finish time $f(a_1) \leq f(c_1)$.
The inductive step, where $i - 1 \to i$. We know that $f(a_{i-1}) \leq f(c_{i-1})$. Let us now consider a_i and c_i. We want to prove that a_i will finish earlier or equal to c_i. By inductive hypothesis, note that c_i is compatible with a_{i-1}. This is because $f(a_{i-1} \leq f(c_{i-1}) \leq s(c_i)$ where s is the start time of c_i. Note that the start time is larger or equal to finish time c_{i-1} because it is compatible to c_{i-1}. We want $f(a_i) < f(c_i)$ because a_i has EFT amongst all jobs that are $a_i \geq a_{i-1}$ and is compatible with a_{i-1}. This concludes the proof. □

3.2 Implementation and Runtime

We now consider the implementation and the runtime. Let us initiate the optimised pseudocode.

```
Sort and renumber jobs s.t. f(1) ≤ f(2) ≤ ... ≤ f(m)
Initiliase set A and we insert {1} since it is already shortest. Let us initialise i to 1.
for j = 2,3,...,m
  if s(j) > f(i), i.e., compatible
  then A ← A ∪ {j}
       i ← j
```

Listing 2: Optimised Pseudo

The running time of this algorithm we will consider $O(n \log n)$ from the most efficient sorting time in step 1. Step 2 is $O(1)$. Step 3 is $O(n-1)$ iterations with $O(1)$ so $O(n-1) \cdot O(1) = O(n)$. Therefore, the running time is $O(n \log n)$.

4 Interval Partitioning / Coloring

```
Input: n open intervals defined as (s_1, f1), (s2, f2), ..., (s_n, f_n)
Output: A coloring κ : [n] → [d] such that the smallest possible number of colors. The idea is
    that all compatible intervals have the same colour.
```

Listing 3: Interval Partitioning

We consider the following definition for this problem:

Definition 4.1. Depth. The depth of a set of intervals is, where $t \in \mathbb{R}$

$$max|\{i \in [n] : t \in (s_i, f_i)\}|$$

which defines the number of intervals that t belongs to.

Theorem 4.1. *The depth is equal to the smallest number of colours in a colouring.*

Proof. We will prove it by creating an algorithm that always uses the depth number of colours to solve the problem. Let us consider the following pseudocode:

```
Consider intervals in some order, which is our greedy rule
If interval i is compatible with some colour c
  we assign κ(i) = c
else1
  we "open" a brand new colour d
  κ(i) ← i
```

Listing 4: Greedy Algorithm for colouring

□

```
Sort lectures s.t. s_1 ≤ s_2 ≤ ... ≤ s_n
d ← 0
for all i = 1, 2, 3, ..., n
  if i is compatible with some classroom c
  then
    κ(i) ← c
  else
    d ← d + 1
    κ(i) ← d
  return κ
```

Listing 5: ESTF greedy algorithm

Proof. Suppose 6 and 7 lines are executed. Then, condition 4 has failed hence i is incompatible with d other lectures. We know that d other lectures start before i (from ESTF greedy rule). Assume all s and f are $\mathbb{Z}$. It follows that if we take $s_i + \frac{1}{2}$, it belongs to $d + 1$ intervals. So depth is at least $d + 1$. □

4.1 Implementation and Runtime

We use a priority queue of classrooms to efficiently perform 4. For classroom c, its key is f_i where i is the latest lecture colored with its c.
The runtime is $O(n \log n) + n \cdot O(\log n) = O(n \log n)$

5 Scheduling to minimise lateness

test

```
Input: n jobs (t_1, d_1), (t_2, d_2), ..., (t_n, d_n) where t is duration and d is duration
Output: Permutation (schedule) j_1, j_2, ..., j_n of 1, 2, ..., n that minimizes maximum lateness
```

Listing 6: The Problem

The lateness l for a job k is calculated by $d_k - f_k$. The maximum lateness would be $max(l_1, l_2, \ldots, l_n)$. We seek to minimise this maximum.

5.1 Implementation and Runtime

We assume that jobs are labelled in the order of their deadlines, that is $d_1 \leq d_2 \leq \ldots \leq d_n$. We will schedule all jobs in this order. Let s be the start time for all jobs. Consider the following algorithm:

```
Order the jobs in order d_1 ≤ ... ≤ d_n
Initially, f = s
For all jobs i = 1, 2, ..., n
  Assign job i to the time interval from s(i) = f to f(i) = f + t_i
  Let f = f + t_i
```

```
38  Return the set of scheduled intervals [s(i), f(i)] for 1, ..., n
```

Listing 7: Implementaiton

Now, we prove this is optimal.

Proof. We will do it through as something we call the exchange argument. Consider the optimal schedule O. Our plan is gradually modify O, preserving its optimality at each step, but eventually transforming it to a schedule that is identical to the schedule A found by the greedy algorithm. Let us define some things:

Definition 5.1. Inversion. We say that a schedule A' has an inversion if a job i with deadline d_i is schedule before another job j with earlier deadline $d_j < d_i$.

We notice that our algorithm has no inversions. However, if there are jobs with identical deadlines there can be many different schedules with no inversions. Regardless, the maximum lateness L is unchanged in such case.

Proof. the maximum lateness L does not change. If two different schedules have neither inversions or idle times (gaps between jobs note: there exists at least one optimal solution in solutions with no gaps), then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Consider such a deadline d. In both schedules, the jobs with deadline d are all scheduled consecutively. Among jobs with deadline d, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. □

We now wish to prove that there is an optimal schedule that has no inversion and no idle time.

Proof. We know that there is an optimal schedule O with no idle time. The proof will consist of a sequence of statements. The first is easy to establish:

If O has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$

We now consider swapping i and j to get rid of the inversion. After swapping, we note that we now have one less inversion. The new swapped schedule has a maximum lateness no larger than that of O. It is clear that if we prove the latest statement, we are done.

Proof. Assume each request r is scheduled for the time interval $[s(r), f(r))$ and has lateness l'_r. Let $L' = \max_r l'_r$ denote the maximum lateness of this schedule. Let $\overline{O}$ denote the swapped schedule; we will use $\overline{s}(r), \overline{f}(r), \overline{l}_r$ and $\overline{L}$ to denote the corresponding quantities in the swapped schedules.
Now recall our two adjacent, inverted jobs i and j. The finishing time of j before the swap is exactly equal to the finishing time of i after the swap. Thus all jobs other than i and j finish at the same time. Moreover, job j will get finish earlier in the new schedule, and hence the swappers does not increase the lateness of job j. Therefore, it is only i that we have to worry about. Its lateness may have been increased, however, its lateness is $\overline{l}_i = \overline{f}(i) - d_i = f(j) - d_i$. The crucial point is that i cannot be more late in the schedule $\overline{O}$ than j was in the schedule O. Specifically, our assumption $d_i > d_j$ implies

$$\overline{l}_i = f(j) - d_i < f(j) - d_j = l'_j$$

Since the lateness of the schedule O was $L' \geq l'_j > \overline{l}_i$, this shows that the swap does not increase the maximum lateness of the schedule. □

Finally, the above proof shows that an optimal schedule with no inversions exists. The proof before that shows all schedules with no inversions have the same maximum lateness, and so the schedule obtained by greedy algorithm is optimal. □

□

6 Shortest Paths in a Graph

Edsger Dijkstra proposed a very simply greedy algorithm to solve the single-source shortest paths problem. The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance $d(u)$ from s, this is the "explored" part of the graph. Initially $S = \{s\}$ and $d(s) = 0$. Now for each node $v \in V - S$, we determine the shortest path can be constructed by traveling along a path through the explored path S to some $u \in S$, followed by a single edge (u, v). That is, we consider the quantity $d'(v) = min_{e=(u,v):u \in S} d(u) + l_e$. We choose the node $v \in V - S$ for which is the quantity minimised, add v to S, and define $d(v)$ to be the value $d'(v)$

```
Let S be the set of explored nodes
  For each u ∈ S, we store a distance d(u)
Initially S = [s] and d(s) = 0
While S ≠ V
  Select a node v ∉ S with at least one edge from S for which d'(v) = min_{e=(u,v):u∈S} d(u) + l_e is a
       small as possible
  Add v to S and define d(v) = d'(v)
```

Listing 8: Dijkstra's Algorithm

In fact this algorithm, for the set S at any point during the execution, for each $u \in S$, the path P_u is a shortest $s - u$ path. This immediately established the correctness of Dijkstra's Algorithm, since we can apply it when the algorithm terminates, at which point S includes all nodes.

Proof. By mathematical induction. Consider the case $|S| = 1$. We have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$. For some value of $k \geq 1$ we now grow S to size $k+1$ by adding the node v. Let (u, v) be the final edge on our $s - v$ path P_v.
By induction hypothesis, P_u is the shortest $s - u$ path for each $u \in S$. Now consider any other $s - v$ path P, we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y. P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S. Indeed, in iteration $k + 1$, Dijkstra's algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favour of adding v. This means that there is no path from s to y through x that is shorter than P_v. But the subpath of P up to y is such a path, and so this subpath is at least as long as P_v. Since edge lengths are nonnegative, the full path P is at least as long as P_v as well. □

6.1 Implementation and Running Time

We can optimise the running time if we use the right data structures. First, we explicitly maintain the values of minima for each node, rather than recomputing them in each iteration. We can further improve efficiency be keeping the nodes $V - S$ in a priority queue with $d'(v)$ as keys. We put nodes V in a priority queue with $d'(v)$ as the key for $v \in V$. To select node v that should be added to set S, we use the extractMin operation. To update the keys, we iterate in which node v is added to S and let $w \notin S$ be a node that remains in the priority queue. Overall, the algorithm can be implemented on a graph with n nodes and m edges in $O(m \log n)$.

7 Minimum Spanning Tree Problem

```
Input: Weighted undirected graph G = (V, E, c : E ↦ ℝ)
Output: A spanning tree of graph G that minimises total cost
```

Listing 9: MST

Definition 7.1. For spanning tree $T = (V_1, F \subset E)$, define $c(T) \stackrel{\text{def}}{=} \sum_{f \in F} c(f)$

A graph has exponentially many spanning trees.

```
Start with empty subgraphy T
Consider edges by increasing cost
Add e ∈ E to T iff T ∪ {e} is acyclic
```

Listing 10: Kruskal's Algorithm

```
Initialise set S = {s} arbitrary s ∈ V
For n − 1 iterations, add S a vertex v ∈ S with smallest cost of attachment to set S such
    that a(v) ≝ min_{w∈S,(v,w)∈S} c([v, w])
Add edge (v, w) to T that is a minimiser in the upper definition.
```

Listing 11: Prim's Algorithm

```
Start with full subgraph T = G
Consider edges by decreasing cost
Delete edge e ∈ E from T iff T − {e} is connected
```

Listing 12: Reverse Deleter

Theorem 7.1. *All* 3 *algorithms are correct. Supplementary assumption: All edge costs are distinct.*

Definition 7.2. A cut set for set of vertices $S \neq$ and $S \not\subseteq$, we define $Cutset(S) = \{(v, w) \in E : v \in S \wedge w \notin S\}$

Fact. (Cutset property). Let $e*$ be a min-cost edge in some $Cutset(S)$. If $T*$ is in MST, then $e* \in T*$ which is equivalent to saying for every MST $T*$, we have $e* \in T*$.

Proof. Proof by contrapositive. If $e* \notin T$ then T is not an MST. We now use exchange argument. Let $e* = (u*, w*)$ and let $e = (u, w)$ for the edge in $Cutset(S)$ in the path from $v*$ to $w*$. Let us consider $(T - \{e\}) \cup \{e*\}$ is a ST (connected, acyclic and spanning). It follows that $c((T - \{e\}) \cup \{e*\}) < c(T)$ from the fact that we have swapped a heavier edge for an edge that weights less □

Corollary. *Kruskal's and Prim's algorithms are correct.*

Fact. The Cycle Property. Let $e \in E$ be the max cost edge in a cycle. If $T*$ is a MST, then $e \notin T*$

Corollary. *All* 3 *algorithms are correct.*

7.1 Implementation and Runtime

```
We use Prioerity queue v ∉ S with a(v) as key
The implementation is close to Dijkstra's algorithm
```

Listing 13: Prim's Algorithm

We obtain $O(m \log n)$ time complexity

```
We sort edges by cost
We then check if it introduces a cycle, we utilise union find data structure
```

Listing 14: Kruskal's Algorithm

Obtaining us another $O(m \log n)$.

8 Divide and Conquer

8.1 Mergesort

1. Divide - split the input into two halves i.e., $\frac{n}{2}$.
2. Recur - solve the two halves RECURSIVELY
3. Conquer - combine results in $O(n)$

Fact. If we denote $T(n)$, the worst case runtime of mergesort MS for all inputs of size n

$$T(n) \leq \begin{cases} O(1) & \text{if } n \leq 2 \\ 2 \cdot T(\frac{n}{2}) + O(n) & \text{otherwise} \end{cases}$$

The above equation is called the mergesort recurrence.

Theorem 8.1. *If n is a power of 2, then $T(n)$ is $O(n \log n)$*

Proof. Recurrence tree analysis. We have $T(n)$. We keep splitting it into a tree from top to bottom, we achieve a tree of depth $\log_2 n$. Therefore, $T(n) = \sum_{n=0}^{\log_2 n} n = O(n \log n)$ □

Proof. Induction.
We guess $T(n) \leq n \log_2 n$. Base case we take $n = 2$. Then $T(n) = 1 \leq 2 \log_2 2$. The inductive step is $\frac{n}{2}$ to n. Assume it holds for $\frac{n}{2}$. We have $T(\frac{n}{2}) \leq 2(\frac{n}{2} \log_2 \frac{n}{2}) + n = n(\log_2 n - 1 + 1) = n \log n$ □

9 Finding the closest pair of points

```
Input: n points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) in a plane
Output: two points such that d(p_i, p_j) is minimal
```

Listing 15: Closest Pair of Points

$$d(p_i, p_j) \stackrel{\text{def}}{=} \sqrt{\left((x_j - x_i)^2 + (y_j - y_i)^2\right)}$$

Fact. Terminal $O(n^2)$ algorithm.
Consider the following figure:

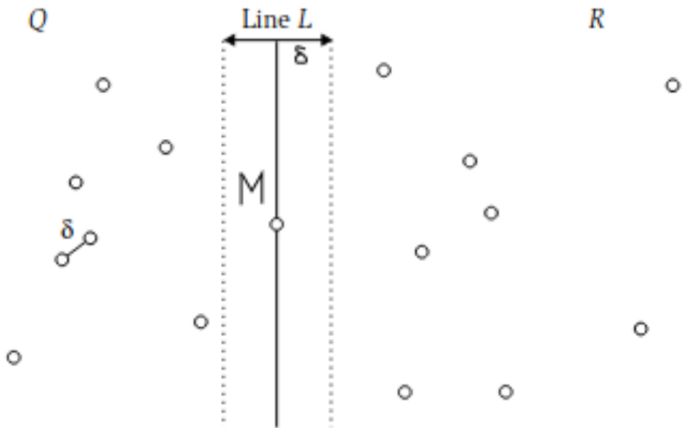


Figure 1: Points in a $2D$ plane

We then find the minimum L to L distance, and similarly, we find the minimum R to R point distance. We also find minimum L to R distance. Call all these distances d_1, d_2, d_3. Then, the minimum distance is $min(d_1, d_2, d_3)$. However, note that, once we find d_1 and d_2, it is sufficient that we only check $\delta \stackrel{\text{def}}{=} min(d_1, d_2)$ to the left and right from the line L, since any larger numbers would mean it is larger than what we have already found.
Fact. For $p_1 \in L$ and $p_2 \in R$, if $(p_1 \in M \vee p_2 \in M)$, then $d(p_1, p_2) \geq \delta$

Corollary. *It suffices to check all points $p_1 \in L$ and $p_2 \in R$ if $p_1 \in M, p_2 \in M$ then $d(p_1, p_2) < \delta$*

Fact. Remainder points of $M = \{p_1, p_2, \ldots, p_k\}$ and $y_1 \leq y_2 \leq \ldots \leq y_k$

10 General recurrence for Divide and Conquer algorithms that are balanced

```
Divide input size A parts of size n/B where B ≥ 2
Solve the A parts of size n/B recursively
Combine results in time O(n^c) for some constant c ≥ 0
```

Listing 16: Algorithm Template

Theorem 10.1. *If*

$$T(n) \leq \begin{cases} O(1) & \text{if } n \leq B \\ A \cdot T\left(\frac{n}{B}\right) + O(n^c) & \text{if } > B \end{cases}$$

then

$$T(n) = \begin{cases} O(n^c) & \text{if } C > \log_B A \\ O(n^{\log_B A}) & \text{if } C < \log_B A \\ O(n^c \cdot \log n) & \text{if } C = \log_B A \end{cases}$$

Proof. Unfolding the recurrence tree. We draw a tree

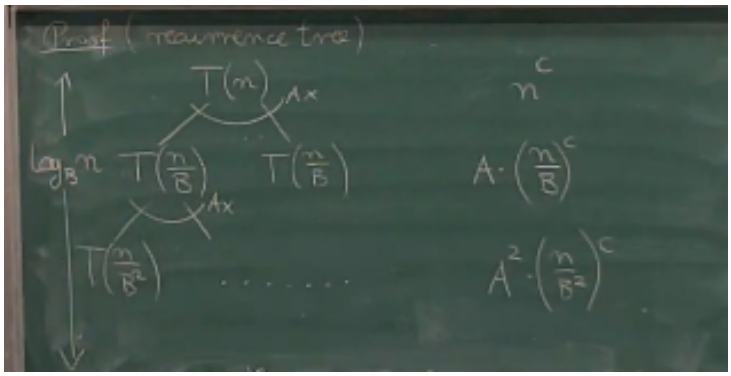


Figure 2: Recurrence Tree

We have $A_i \left(\frac{n}{B_i}\right)^c$ at level i, which can be simplified to $\left(\frac{A}{B^C}\right)^i \cdot n^c$ Then,

$$T(n) = O(n^c \cdot \sum_{i=0}^{\log_B n} \left(\frac{A}{B^c}\right)^i$$

If $A = B^c$ we have $\log_B A = \log_B(B^{c)} = c$. Then,

$$\sum_{i=0}^{\log_B n} \left(\frac{A}{B^c}\right)^i = \sum_{i=0}^{\log_B n} 1 = O(\log_B n)$$

Therefore we have

$$T(n) = O(n^c \log n)$$

If we have $A < B^c (\log_B A < c)$, then

$$\sum_{i=0}^{\log_B A} = O(1)$$

from the fact that it is a geometric series. It follows that then $T(n) = O(n^c)$.
If $A > B^c (\log_B A > c)$, then

$$\sum_{i=0}^{\log_B n} \frac{\left(\frac{A}{B^c}\right)^{\log_B n} - 1}{\frac{A}{B^c} - 1}$$

The denominator is a constant, therefore we obtain

$$\begin{aligned} &O(\frac{A}{B^c \cdot \left(\frac{A}{B^c}\right)^{\log_B n}} \\ =&O(\frac{B^{\log_B A \cdot \log_B n}}{B^{c \cdot \log_B n}}) \\ =&O(\frac{n^{\log_B A}}{n^c}) \\ =&O(n^{\log_B A}) \end{aligned}$$

Which implies $T(n) = O(n^{\log_B A})$ □

11 Integer Multiplication

Integer multiplication in computer science what may seem to be a simple problem, is actually not so simple. We will assume the multiplication is done in base 2, although the end result does not change. We begin writing x as

$$x = x_1 \cdot 2^{\frac{n}{2}} + x_0$$

Therefore, for the problem xy we have

$$\begin{aligned} xy &= (x_1 \cdot 2^{\frac{n}{2}} + x_0)(y_1 \cdot 2^{\frac{n}{2}} + y_0) \\ &= x_1 y_1 + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0 \end{aligned}$$

This reduces the problem of solving a single n bit problem instance to the problem of solving four $\frac{n}{2}$ bit instances. So we have a first candidate for a divide and conquer solution: recursively compute the results for these four $\frac{n}{2}$ bit sequences and then combine them as we have done above. Combination requires only addition, which is $O(n)$. We obtain the solution $T(n) \leq O(n^{\log_2 q} = O(n^2)$. Our divide and conquer algorithm with four branching was a complicated to get back to quadratic time, if we have $q = 4$. Therefore we should focus on obtaining $q = 3$, i.e., try to get away with three recursive calls, resulting in $T(n) \leq O(n^{\log_2 q}) = O(n^{1.59})$.
Recall that our goal is to compute the expression $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0$. To make it three recursive calls, consider the result of the multiplication $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + y_1 x_0 + x_0 y_0$. This has the four products above added together, at the cost of a single recursive algorithm. If we determine $x_1 y_1$ and $x_0 y_0$ by recursion, we get the outermost terms explicitly, and we get the middle term by subtracting $x_1 y_1$ and $x_0 y_0$ away from $(x_1 + x_0)(y_1 + y_0)$. Therefore our algorithm is

```
    x = x_1 · 2^(n/2) + x_0
y_1 · 2^(n/2) + y_0
    compute x_1 + x_0 and y_1 + y_0
    p = recursiveMultiply x_1 + x_0, y_1 + y_0
x_1 y_1 = recursiveMultiply(x_1, y_1)
x_0 y_0 = recursiveMultiply(x_0, y_0)
    Return x_1 y_1 · 2^n + (p − x_1 y_1 − x_0 y_0) + 2^(n/2) + x_0 y_0
```

Listing 17: Recursive Multiply

https://www.youtube.com/watch?v=JCbZayFr9RE

$$T(n) = \begin{cases} O(1) & \text{if } x, y \text{ are single digits, i.e., } n = 1 \\ 3T\left(\frac{n}{2}\right) + O(n) & \text{otherwise} \end{cases}$$

Note that $A = 3$ because there are 3 recursively calls, $c = 1$ since we combine them using only addition and subtraction, $B = 2$ since we split the number into two. Therefore, we obtain asymptotic analysis of, for the case $\log_2 3 > 1$:

$$O(n^{\log_2 3})$$

12 Subset Sum and Knapsack

```
Inputs: Positive integers w_1, w_2, ..., w_n, and W
Output: Subset S ⊂ {1, 2, ..., n} s.t. max(Σ_{i∈S} w_i) with constraint Σ_{i∈S} w_i < W
```

Listing 18: Subset Sum

Note that no correct and efficient greedy algorithms are known for such problems. We will solve this using dynamic programming and breaking it down into sub problems.

12.1 Sub problems

We can consider items from 1 to j where $j = 0, 1, \ldots, n$. We can come up with a suitable recurrence.

Definition 12.1.

$$M(j) \stackrel{\text{def}}{=} \max\left(\sum_{i \in S} w_i \text{ s.t. } S \subseteq \{1, 2, \ldots, j\} \wedge \sum_{i \in S} w_i \leq W\right)$$

Fact. If j is not in an optimal solution S, then $M(j) = M(j-1)$
Fact. If j is an optimal solution S, then $M(j) = w_j + M(j-1)$ However, note that this fact is not true, as we do not consider the change in W, the weight. Therefore, this is subproblem does not work. Let us consider another one instead.
Instead, let us consider pairs that is $(\{1, 2, \ldots, j\}, C)$ where $j = 0, 1, \ldots, n$ and $C = 0, 1, \ldots, W-1, W$. We can now adjust our facts.

Definition 12.2.

$$M(j, C) \stackrel{\text{def}}{=} \max\left(\sum_{i \in S} w_i \text{ s.t. } S \subseteq \{1, 2, \ldots, j\} \wedge \sum_{i \in S} w_i \leq C\right)$$

We can now rewrite our last fact.
Fact. If j is in an optimal solution S, then $M(j, C) = w_j + M(j-1, C-w_j)$
Fact. $M(j, C)$ satisfies the following recurrence

$$M(j, C) = \begin{cases} 0 & \text{if } j = 0 \wedge C = 0 \\ M(j-1, C) & \text{if } w_j > C \\ \max\left(M\left(j-1, c\right), w_j + M(j-1, c-w_j)\right) & \text{if } w_j \leq C \end{cases}$$

12.2 Compute

```
for j = 0, 1, 2, ..., n do
  for C = 0, 1, 2, ..., W do
    compute the recurrence described above
```

Listing 19: Computing our algorithm

If W is represented in binary/decimal, we have size of representation of W as $\log_2 W$. However, W is $\Omega\left(2^{\text{size of representation of } W}\right)$.
We have runtime $O(n \cdot W)$

13 Sequence Alignment

One way is to group up, for examples, a sequence such as letters. For example,

Table 1: Sequence Alignment

O	C	U	R	R	A	N	C	E	-
O	C	C	U	R	R	E	N	C	E

However, with this table we have many letter mismatches. If we modify it as such:

Table 2: Sequence Alignment Redesign

O	C	-	U	R	R	A	N	C	E
O	C	C	U	R	R	E	N	C	E

Definition 13.1. Levenshtein, 1966.

$$\Delta(x, y) \stackrel{\text{def}}{=} \min \text{Cost}(A)$$

where x, y are letters and A are different alignments of x and y.

Definition 13.2. Cost. Cost of an alignment is defined to be the sum of all gap and mismatch penalties.

Consider δ which is gap penalty, and further, consider $\alpha_{ab \in \Sigma}$. Let $x, y \in \Sigma^*$
Now, let $\delta = 2$ and

$$\alpha_{ab} = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if both } a \text{ and } b \text{ are both vowels or consonants} \\ 3 & \text{otherwise} \end{cases}$$

```
77  Input: δ, α_ab penalties. Two words x, y with x_1, ..., x_m, y_1, ..., y_n ∈ Σ* words respectively
78  Output: Δ(x, y), an alignment of A ∈ Alignments(x, y) of min Δ(x, y)
```

Listing 20: Algorithm

13.1 Subproblems

Fact. In an optimal alignment of x and y, one of the following three cases holds:

1. Last letters of x and y are matched
2. Last letter of x is aligned with a gap
3. Last letter of y is aligned with a gap

Definition 13.3. The subproblems and their best solution values. For all $i = 0, 1, \ldots, m$ and for all $j = 0, 1, \ldots, n$ we define

$$D(i, j) \stackrel{\text{def}}{=} \Delta([x_1, x_2, \ldots, x_i], [y_1, y_2, \ldots, y_j])$$

Fact. Recurrence. $D(i, j)$ satisfies the following recurrence:

$$D(i, j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min\left(\alpha_{x_i, y_j} + D(i-1, j-1)\right) & \text{if } i \geq 1 \wedge j \geq 1 \\ \delta + D(i-1, j) & \text{if } x_i \text{ is aligned with a gap} \\ \delta + D(i, j-1) & \text{if } y_j \text{ is aligned with a gap} \end{cases}$$

It follows that the complexity of the implementation is $O(mn)$. The space complexity is $O(m)$. However, to use linear space, we need to make observations:
Fact. The solution to finding linear space is to realise that it is equivalent to understanding the bottom diagram shortest path problem.

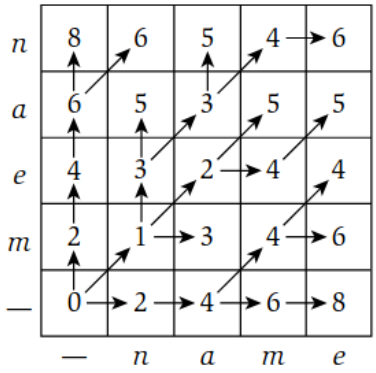


Figure 3: NAME array

Notice that if we use horizontal edges or vertical edges it means that we use a gap. So all horizontal and vertical edges have cost 2. α_{ab} denotes the weight of the edge that are slanted, depending on the letters.
Fact. A min-cost alignment is a shortest path from $(0, 0)$ to (m, n) in the graph above.Therefore, it makes sense to find the shortest path to each vertex, and then work backwards. By doing so, we find that the best alignment is

Table 3: MEAN-NAME alignment

M	E	A	N	-
N	-	A	M	E

Now notice that the maximum amount of edges we need to store is $O(m + n)$.

Definition 13.4.

$$D(i, j) \stackrel{\text{def}}{=} \Delta([x_1, x_2, \ldots, x_i], [y_1, y_2, \ldots, y_j])$$
$$E(i, j) \stackrel{\text{def}}{=} \Delta([x_{i+1}, x_{i+2}, \ldots, x_m], [y_{j+1}, y_{j+2}, \ldots, y_n])$$

Fact. $D(i, j)$ is the cost of the shortest path from $(0, 0)$ to (i, j) and $E(i, j)$ is the length of the shortest path from (i, j) to (m, n).
Fact. The length of a shortest $(0, 0)$ to (m, n) path that passes through some $(\bar{i}, \bar{j})$ is $D(\bar{i}, \bar{j}) + E(\bar{i}, \bar{j})$.
If we fix some $\bar{j}$ that is arbitrary. For $\bar{j} \in \{0, 1, \ldots, n\}$, if $\bar{i} \in \{0, 1, \ldots, m\}$ minimizes the sum in the above fact, we have found the method to do a recurrence.

```
Input: two strings x, y
If x_n = 2 ∨ y_m = 2
then return algorithm naive alignment of x and y
else compute D([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_{m/2}])
  compute E([x_1, x_2, ..., x_n], [y_{m/2+1}, ..., m])
  find i that minimizes D(i, n/2) + E(i, n/2)
  add (i, n/2) to P (path)
  call divide-and-conquer alignment to ith prefix of x and n/2 prefix of y
  call divide-and-conquer alignment to m from i + 1th prefix x and n from n/2 + 1th prefix of y
```

Listing 21: Divide and conquer alignment algorithm

13.2 Runtime Recurrence

Consider $T(n,m) \leq T(m-q, \frac{n}{2}) + T(q, \frac{n}{2}) + O(nm)$, we get the following tree:

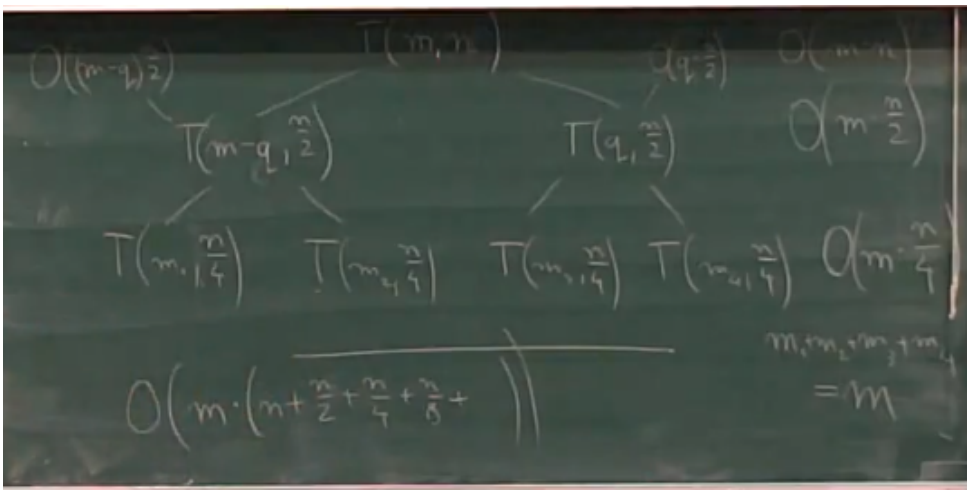


Figure 4: Tree Recurrence

Notice that we get consistently m but a geometric series n if we sum all levels, that is,

$$\begin{aligned} & O(m \cdot (n + \frac{n}{2} + \frac{n}{4} + \ldots)) \\ =& O(m \cdot n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i}) \\ =& O(2mn) \\ =& O(mn) \end{aligned}$$

14 Revisit to Shortest Paths

```
Inputs: Directed graph with edges costs G = (V, E|c : E ↦ ℤ) and target vertex t ∈ V
Output: If G has no negative cycles, then output for every v ∈ V,
  output the min cost of v − t path
  output the path itself
```

Listing 22: Single-Target-Shortest-Paths

```
Input: Graph with edge costs G = (V, E, c : E ↦ ℤ)
Output: Yes if G has a negative cost cycle, no otherwise
```

Listing 23: Negative Cycle

Fact. From week 3, Dijsktra fails.
Fact. If G has no negative cycles, then there is min-cost $v - t$ path that is simple.

Proof. Let path P be a min cost $v - t$ path with a cycle. Let us call the path $v - w$ the path p' and $w - t$ the path p''. Assume a cycle c exists in w. Notice that we can take the path p' and p'' is also a $v - t$ path. If we do not have negative edges, then

$$c(p' + p'') = c(p') + c(p'') \leq c + (p') + c(c) + c(p'')$$

□

14.1 Subproblems

$$D(i,v) \stackrel{\text{def}}{=} min\,(c(P) : v-t \text{ path } \wedge P \text{ has } \leq i \text{ edges })$$

Note that $i \in \{0,1,\ldots,n-1\}$ and $v \in V$.
Fact. If P is a mincost $v-t$ path with at most i edges and $i > 0$, then if P has less than $i-1$ edges, then $c(P) = D(i-1,v)$. If P has exactly i edges and $(v,w) \in E$ and is the first edge in D, then the cost of the shortest path that uses at most i edges and goes from $v-t$ is equal to the cost of that first edge and the optimal path from $c(v,w) + D(i-1,v)$
Fact. The following recurrence holds:

$$D(i,v) = \begin{cases} 0 & \text{if } i = 0, v = t \\ +\infty & \text{if } i = 0, v \neq t \\ min(D[i-1,v], min_{(v,w)\in E}(c(w) + D(i-1,v))) & \text{if } i > 0 \end{cases}$$

14.2 Compute

```
Evaluate D(i,v) for i = 0,...,n-1
Return the array (D(n-1,v))_{v∈V}
```

Listing 24: Shortest Path(G,t)

Fact. Shortest-Paths-Algorithm returns mincosts of $v-t$ paths for all $v \in V$ and runs in time $O(nm)$ and quadratic space $O(n^2)$

14.3 Bellman-Ford

2

```
Initialise BF(v) that is 0 if v = t, +∞ otherwise
for i = 0,1,...,n-1
  for v ∈ V do
    BF(v) ← min(BF(v), min_{(v,w)∈E}(c(v,w) + BF(w)))
Return BF(v) for all vertices v
```

Listing 25: The Algorithm

Fact. Correctness and complexity. At all times, $BF(v) = c(P)$ for some $v-t$ path.
Fact. After iteration i, the value of $BF(v) \leq D(i,v)$

14.4 Finding Mincost Paths

Value $BF(u)$ updated to $(c(u,w) + BF(w))$ for some $(v,w) \in E$. Let $first(v) \leftarrow w$.
Fact. If G has no negative cycles, then when the Bellman-Ford algorithm terminates, got all $v \in V$, the path V_0, $(v, first[V_0])$, which we call V_1, then $(first[V_0], first[V_1]$ and so on, we eventually arrive at t.
Fact. Every cycle in the pointer graph is negative.

Proof. Along edges $(v, first[v])$ we have $BF(v) \geq c(v,w) + BF(w)$. Now consider (v,w), which is the last edge to a cycle C of the form $v_1, v_2, \ldots, v_k$ in the pointer graph. Just before that, we must have had that this inequality is strict, i.e., $>$ instead of $\geq$

$$\sum_{v\in C} BF(v) > \sum_{v\in C} c(v, first(v)) + BF(first(v))$$
$$0 > c(C)$$

□

15 From negative cycles to sink

```
Input: directed graphs G(V, E, c : E ↦ ℤ) and t ∈ V
Output: A negative cycle with a path to t
```

Listing 26: Algorithm

Fact. The following "algorithm" solves this problem

```
Input G = (V, E, c : E ↦ ℤ)
Construct G' from G by adding t to V' and for all v ∈ V add an edge (v, t) of cost 0
Return from next cycle to sink
```

Listing 27: A

Fact. G has no negative cycle with a path to t if and only if for all $v \in V$, $D(n-1, v) = D(n, v)$

16 Intractability

We must classify all algorithmic problems.

$$\text{TRACTABLE} \stackrel{\text{def}}{=} \text{Polynomial Time Solvable}$$

The negation of the above statement is Intractable. We say that Tractable are P. However, there are algorithms for which we cannot prove tractable or intractable, we call these the gray zone algorithms. Some examples are Knapsack and Subset Sum. The gray area has a huge amount of classifications, however, for this module we focus only on $NP - complete$.
Lots of natural and important algorithmic problems have been classified as "equally hard/easy/complex", defined as $NP - complete$. To classify these problems, we use the tool called **Polynomial-time Reduction**. Polynomial time reduction allows us to compare relative complexity of algorithmic problems and has 2 key uses:

1. Algorithm Design
2. Proving Hardness

17 Polynomial-Time Reduction

Definition 17.1. If X is an algorithmic problem, then an instance of X is simply a legal input to X.

Definition 17.2. Algorithms with oracles. For Y an AP, a Y-oracle Call is a "basic operation" that can

1. read an instance y of Y from a designated space in memory
2. outputs a correct output $Y(y)$ in a designated space in memory
3. It works in time linear in $|y| + |Y(y)|$

Definition 17.3. Let X and Y be algorithmic problems. A polynomial-time reduction from X to Y is a polynomial-time algorithm for X that can make Y-oracle calls.

We write $X \leq_p Y$ if there exists a poly-time reduction from X to Y.

Example 17.1. A linear-time reduction from MULTIPLY to SQUARE

```
AB ← SQUARE(a + b)
ab ← SQUARE(a − b)
output (AB − ab)/4
```

Listing 28: MiltiplyBySquaring

Fact. Algorithm Design using reductions. Suppose $X \leq_p Y$. If $Y \in P$, then $x \in P$.

Proof. Let A be a poly-time reduction from X to Y. Let B be a poly-time algorithm for solving instances of Y. Let algo A where all $Y \leftarrow B$ be obtained from A by replacing Y-oracle calls by copies of B. The Runtime $T_{A(Y \leftarrow B)}(n) \leq T_A(n) \cdot T_B(T_A(n))$. That is, from the fact that the maximum input we can put into algo B is the runtime of A. □

We can also restate fact 5 by its negation. That is, if $X \notin P$, then $Y \notin P$. This means that the same fact shows us that

$$\underbrace{X}_{\text{no harder than } X} \quad \leq_p \quad \underbrace{Y}_{\text{no easier computationally than } X}$$

18 Independent Set

```
Inputs: Undirected graph G = (V, E), number k
Output: Yes if G has an independent set of size k, no otherwise
```

Listing 29: Algorithm

Definition 18.1. $S \subseteq V$ is independent if for all $v, w \in S$, we have $(v, w) \notin E$, i.e., they're not connected by edges

19 Vertex Cover

```
Inputs: Undirected graph G = (V, E), number k
Output: Yes if G has a vertex cover of size k, no otherwise
```

Listing 30: Algorithm

Definition 19.1. $S \subseteq V$ is a vertex cover in G if the following condition holds. For all edges $(v, w) \in E$, $v \in S$8 or $w \in S$. Vertices cover edges.

Question: Is Independent Set $\leq_p$ Vertex Covering and Vertex Covering $\leq_p$ Independent Set. To answer, observe that

$$IS \leq_p \text{MaxSize of } IS$$

where IS is independent set. Furthermore,

$$IS \geq_p \text{MaxSize of } IS$$
$$IS \geq_p \text{A MaxSize of } IS$$

Fact. In a graph G, S is an IS iff $V - S$ is a vertex cover

Proof. If $v \in S$ and $w \in S$, then $(w, v) \notin S$. The negation is if $(w, v) \in S$, then $v \notin S$ or $w \notin S$. This is equivalent to saying if $(w, v) \in E$, then $w \in V - S$ or $v \in V - S$. □

Fact.

$$IS \leq_p VC$$
$$VC \leq_p IS$$

Proof. We have the following poly-time reduction from IS to VC with algorithm A of input graph G and a number k.

$$A(G, k) = VC(G, n - k)$$

□

20 Set Cover

Input: Set U of n elements, a collection of subsets $s_1, s_2, \ldots, s_m$ and a positive integer k
Output: If there is a collection of k number of sets, such that the union of all chosen subsets are equal to U.
Fact:

$$VC \leq_p SC$$

Definition 20.1. For graph $G = (V, E)$ and $u \in V$, define $I_u = \{v \in V : u \text{ is in pair with } v\}$

Fact. $C \subset V$ is a vertex cover in G if and only if C is a set cover in E, T

Proof. C is a set cover in $E; T_n$ where T_n is a family of subsets if and only if for all edges $e = (v, w) \in E$ there is some $u \in C$ such that $e \in I_u$. This simply means $v \in C$ or $w \in C$. It means that C is a vertex covering, as it covers v or w. □

Proof of the fact the poly-time reduction:

$$A((V, E), k) = SC(E, V, k)$$

By fact above, SC is a yes instance of VC if and only if for all vertices I_u is a yes instance.

21 SAT

Input: Boolean expressions ϕ in conjunctive normal form over a set of variables $X = (x_1, x_2, \ldots, x_n)$
Output: Yes if and only if there is a truth assignment $V : X \mapsto \{0, 1\}$ that makes expression ϕ true i.e., ϕ is satisfiable.

Definition 21.1. Literal is a variable x_i or its negation.

Definition 21.2. Clause is a disjunction of literals, e.g., $l_1 \vee l_2 \vee \ldots \vee l_k$

Definition 21.3. Disjunction Normal Form is a disjunction of clauses

21.1 3-SAT

3-SAT is similar to SAT, but we restrict our inputs that in each clause we have at most 3 literals.

Theorem 21.1. *There is a polynomial time reduction from 3-SAT to Independent Set i.e.,*

$$3 - SAT \leq_p IS$$

We pick one of the three literals occurrence in that clause and set it to 1. *While we do this we need to ensure that we avoid conflicts.*