Design and Analysis of Algorithms

Joshua Clark

March 29, 2013

Contents

List of Algorithms

1 Design Principles

1.1 Divide and Conquer

The divide-and-conquer strategy can be thought of as solving problems in the following steps:

1. Break the initial problem into subproblems.
2. Recursively solve the subproblems [if the problems are small enough, solve by brute force for a *base case*].
3. Appropriately combine the answers to the subproblems.

The most complicate work is found in dividing the problems into the subproblems, at the tail-end of the recursion, when solving the subproblems, and gluing the intermediate answers together.

1.1.1 Merge Sort

Description This is a divide-and-conquer algorithm to sort the array $A\,[p..r)$.

1. *Divide* – split the array into $A\,[p..q)$ and $A\,[q..r)$, where $q = \lfloor (p+q)/2 \rfloor$.
2. *Conquer* – by recursively sorting the subarrays, and bottoming out the recursion when singleton arrays are reached.
3. *Combine* – by merging the sorted subarrays $A\,[p..q)$ and $A\,[q..r)$ using a $\Theta(n)$ procedure.

Algorithm 1: Merge Sort

```
def MERGE-SORT ($A, p, r$)
    Input: An integer array $A$ with indices $p < r$
    Output: The
            subarray $A[p..r)$ sorted in increasing order
    if $p + 1 < r$ then
        $q = \lfloor (p+r)/2 \rfloor$;
        MERGE-SORT ($A, p, q$);
        MERGE-SORT ($A, q, r$);
        MERGE ($A, p, q, r$);

def MERGE ($A, p, q, r$)
    Input: Array
           $A$ with indices $p, q, r$ such that $p < q < r$ and
           subarrays $A[p..q)$ and $A[q..r)$ already sorted.
    Output: The subarrays
            are merged into a sorted array $A[p..r)$
    $n_1 = q - p$, $n_2 = r - q$;
    Create array $L$ of size $n_1+1$, Create array $R$ of size $n_2+1$;
    for $i = 0$ to $n_1 - 1$ do $L[i] = A[p+i]$;
    for $j = 0$ to $n_2 - 1$ do $R[j] = A[q+j]$;
    $L[n_1] = \infty$, $R[n_2] = \infty$;
    $i = 0$, $j = 0$;
    for $k = p$ to $r - 1$ do
        if $L[i] \le R[j]$ /* To ensure stable sort */
        then $A[k] = L[i]$; $i = i + 1$;
        else $A[k] = R[j]$; $j = j + 1$;
    return $A$;
```

Pseudo-code `MERGE` runs in $\Theta(n)$ as each of the `for` loops' lines run in constant time, iterating over n elements, therefore the entire function is bounded by n.

Advantages

- Merge sort is stable, as the `MERGE` function is left biased.
- Runtime is always $O(n \log n)$.

Disadvantages

- Merge, and by extension, merge sort requires $O(n)$ extra space.
- Merge sort is not an online algorithm.

Haskell Implementation By pattern matching, first against the base cases, the program then recurses into two more calls of mergesort using two separate halves of the list. The call to merge then ensure that the two halves of the list are joined into a correctly sorted list.

```
mergesort :: Ord a ⇒ [a] → [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort xs1) (mergesort xs2)
        where (xs1,xs2) = split xs

split :: [a] → ([a],[a])
split (x:y:zs) = (x:xs,y:ys) where (xs,ys) = split zs
split xs = (xs,[])

merge :: [a] → [a] → [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) =
  case x ≤ y of
        True → x : merge xs (y:ys)
        False → y : merge (x:xs) ys
```

1.1.2 Binary Search

Description and Requirements

- The array to work with is sorted
- Begin by looking at the middle element. If the item to be found is less than the middle element, discard the latter half of the list, else discard the first half.
- Recursively look for the element until a singleton is reached. If this is the element, then return true/index of element, else return false/sensible error variable.

Algorithm 2: Binary Search

```
Input: Array
       A of distinct increasing integers, and an integer $z$
Output: "Yes" if $z \in A$, "No" otherwise
1  def BIN-SEARCH $(A, p, r, z)$
2      if $p \geq r$ then
3          return "No"
4      else
5          $q = \lfloor (p+q)/2 \rfloor$;
6          if $z = A[q]$ then
7              return "Yes"
8          else
9              if $z < A[q]$ then
10                 BIN-SEARCH$(A, p, q, z)$;
11             else
12                 BIN-SEARCH$(A, q+1, r, z)$;
```

Running time Let $T(n)$ be the worst-case time, therefore

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + O(1) & \text{otherwise} \end{cases}$$

By the Master Theorem, $T(n) = O(\log n)$. By looking at the "decision tree" of the binary search, it is clear that binary search is $\Omega(\log n)$, therefore, $T(n) \in \Theta(\log n)$.

1.1.3 Selection Problem

The i^{th} ordered statistic of a set of n distinct elements is the element that is larger than exactly $i-1$ other elements.

Upper-bound time: $O(n \log n)$ – sort the array in $O(n \log n)$ time and return the i^{th} element.

It is possible to do so in worst case $O(n)$ time.

By partitioning the array, it is possible to recursively solve the above problem.

Algorithm 3: Partition algorithm

```
Input: An array $A$ of distinct numbers,
       with indices $p \leq q < r$ and $m = A[q]$ as the pivot
Output: An index $q'$ with $p \leq q' \leq r$ such that
        $A[p..r)$ is a permutation of $A$, $\forall a \in A[p..q') \Rightarrow$
        $a < m \wedge \forall a \in A[q'..r) \Rightarrow a \geq m \wedge A[q'] = m.$
```

Algorithm 4: Selection Algorithm

```
Input: An array $A$ of
       distinct numbers and the $i^{\text{th}}$ order statistic to find
Output: The $i^{\text{th}}$ smallest element
1  Divide the $n$ input elements
   into groups of 5, with one group of $n \mod 5$ elements;
2  Find the median of the first $\lfloor n/5 \rfloor$ groups in $O(1)$;
3  Find the median-of-medians $x$ by calling select recursively;
4  Partition the input array
   around $x$, with the lower partition having $k-1$ elements;
5  if $i = k$ then return $x$;
6  else
7      if $i < k$ then SELECT$(A[0..k), i)$;
8      else SELECT$(A[k+1..n), i-k)$;
```

Steps 1,2, and 4 take $O(n)$ time.

The number of elements $\leq x$ is, at least $3\left(\lceil \frac{1}{2} \lfloor \frac{n}{5} \rfloor \rceil\right) \geq \lceil \frac{3n}{10} \rceil - 2$. Thus, in the worst case, SELECT is called on, at most $\lfloor 7n/10 \rfloor + 2$ elements, so

$$T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor + 2) + cn$$

for some constant $c \in \mathbb{R}$.

By supposing that $T(n) \in O(n)$, and then substituting bn in for $T(n)$, for $b \geq 10cn/(n-20)$, $T(n) \leq cn$ else $T(n) \in O(1)$.

1.1.4 Integer Multiplication

By using an observation of Gauss', that is $(a+bi)(c+di) = ac - bd + (bc+ad)i$ can be done using just three multiplication operations (ac, bd, and $(a+b)(c+d)$), integer multiplication can then be performed much more efficiently. By splitting n-bit numbers x and y into left and right halves, each $n/2$ bits long, it is therefore possible to compute

$$\begin{aligned} xy &= \left(2^{n/2} x_L + x_R\right)\left(2^{n/2} y_L + y_R\right) \\ &= 2^n x_L y_L + 2^{n/2}\left(x_L y_R + x_R y_L\right) + x_R y_R \end{aligned}$$

Multiplication by 2^n can be implemented as a left-shift in constant time, and addition in linear time, giving the time $T(n)$ to multiply two n-bit numbers as

$$T(n) = 4T(n/2) + O(n)$$

which, by the Master Theorem, gives $T(n) \in O(n^2)$.

By using Gauss' trick, and re-writing as $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$, the running time becomes

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

Again, by the Master Theorem, $T(n) \in \left(n^{\log_2 3}\right) \approx O\left(n^{1.59\ldots}\right)$.

1.1.5 Matrix Multiplication

Multiplying the $p \times q$ matrix X by the $q \times r$ matrix Y gives

$$Z_{ij} = \sum_{k=1}^{q} X_{ik} \cdot Y_{kj}$$

The above requires $p \times q \times r$ multiplications and $p \times (q-1) \times r$ additions. When $p = q = r = n$, there are $2n^3 - n^2 \in O(n^3)$ operations.

By dividing X and Y into quarters,

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

It is then possible to recursively multiply the smaller matrices, and then add the individual elements in $O(n^2)$ time, giving a total running time of $T(n) = 8T(n/2) + O(n^2)$, which, by the Master Theorem, is $O(n^3)$.

By observing (similarly to Gauss) that

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Thus giving seven multiplications, hence $T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) \in O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right)$.

1.2 Dynamic Programming

By identifying and solving a collection of smaller subproblems, and initial problem can be solved by using the solution to the smaller problems in building up a complete solutions. Dynamic programming is not suited to solving all problems, as the solutions to the subproblems are not necessarily *compositional*, i.e. the optimal solution is not formed from a composite of the solutions of the subproblems.

Dynamic programming is only applicable when the *Principle of Optimality* is satisfied: "The optimal solution to a problem is a composite of optimal solutions to (some of) its subproblems".

1.2.1 Knapsack Problems with Repetition

For the n items, each of weight $w_1, w_2 \dots w_n$ and value $v_1, v_2, \dots v_n$ respectively, and a maximum carrying capacity of W find the maximum possible value to carry without the total weight exceeding W. Define $K[w]$ to be the maximum achievable value with weight limit w. The solution to the problem is therefore

$$K[w] = \max \{K[w - w_i] + v_i : w_i \leq w\}$$

where $K[W]$ is the desired answer.

The above fills an one-dimensional array of length $W + 1$, each entry taking $O(n)$ time, hence the total running time of the algorithm is $O(nW)$.

1.2.2 Knapsack Problems without Repetition

For the n items, each of weight $w_1, w_2 \dots w_n$ and value $v_1, v_2, \dots v_n$ respectively, and a maximum carrying capacity of W find the maximum possible value to carry without the total weight exceeding W, and with all items being distinct. Define $K[w,]$ to be the maximum achievable value with weight limit w, choosing from items $1, 2 \dots j$, as j varies between $0 \leq j \leq n$. The solution to the problem is therefore

$$K[w, j] = \max \{K[w - w_j, j-1] + v_j, K[w, j-1]\}$$

where $K[W, n]$ is the desired answer.

Algorithm 5: Knapsack Problem with repetition

```
Input: List of weights $\{w_1, w_2, \dots, w_n\}$ and respective
       values $\{v_1, v_2, \dots, v_n\}$, and maximum weight $W$
Output: Maximum value of items
1 $K[0] = 0$;
2 for $w = 1$
  to $W$ do $K[w] = \max \{K[w - w_i] + v_i : w_i \leq w\}$;
3 return $K[W]$;
```

Algorithm 6: Knapsack Problem without repetition

```
Input: List of weights $\{w_1, w_2, \dots, w_n\}$ and respective
       values $\{v_1, v_2, \dots, v_n\}$, and maximum weight $W$
Output: Maximum value of items
1 for $j = 0$ to $n$ do $K[0, j] = 0$;
2 for $w = 0$ to $W$ do $K[w, 0] = 0$;
3 for $j = 1$ to $n$ do
4     for $w = 1$ to $W$ do
5         if $w_j > w$ then
6             $K[w, j] = K[w, j-1]$;
7         else
8             $K[w] = \max \{K[w - w_i] + v_i : w_i \leq w\}$
9 return $K[W, n]$;
```

The above fills an one-dimensional array of length $W + 1$, each entry taking $O(n)$ time, hence the total running time of the algorithm is $O(nW)$.

```
knapsack_spec
     :: [(Integer,Integer)] → Integer → Integer
knapsack_spec wvs w =
      maximum ( map (sum ∘ map snd) (
      filter ((≤ w) ∘ sum ∘ map fst) (
      subsequences wvs )))

subsequences :: [a] → [[a]]
subsequences = foldr f [[]]
  where f x = foldr (λy zs → (x:y):y:zs) []

knapsack_rec [] w = 0
knapsack_rec ((wi,vi):wvs) w
  | wi > w = knapsack_rec wvs w
  | otherwise = max (knapsack_rec
       wvs w) (knapsack_rec wvs (w-wi) + vi)

knapsack_dp wvs wtot = table ! (wtot,n)
  where n = length wvs
        table = array ((0,0),(wtot,n)) [((w,
            j), ks w j) | w ← [0..wtot], j ← [0..n]]
        ks w 0 = 0
        kw w j = if wj > w then table (w,j-1)
                     else max (table ! w,j-1)
                       ((table ! (w-wj, j-1)) + vj)
          where (wj,vj) = wvs !! (j-1)
```

1.2.3 Change Making Problem

Assuming an unlimited supply of coins, what is the minimum number of coins needed to give change to value v using denominations $1 = x_1, x_2, \ldots, x_n$?

By setting $C[u]$ to be the minimum number of coins required to give change to a total value of u, and looking for $C[v]$, the following recurrence is constructed:

$$\begin{aligned} &C[0] = 0 \\ &C[u] = 1 + \min\{C[u - x_1] : 1 \le i \le n \wedge u \ge x_i\} \end{aligned}$$

The above fills an one-dimensional array of length v with each entry taking, at most $O(n)$ time, hence the total running time is $O(nv)$.

Algorithm 7: Change giving algorithm

```
Input: List of coin denominations
       $1 = x_1, x_2, \ldots, x_n$, and value of change $v$
Output: Minimum number of coins required to give change
1 $C[0] = 0$;
2 for $u = 1$ to $v$ do
3     $C[u] = 1 + \min\{C[u - x_1] : 1 \le i \le n \wedge u \ge x_i\}$;
4 return $C[v]$;
```

1.2.4 Edit Distance Problem (Levenshtein Distance)

The edit distance of a string is the "minimum number of edits needed to transform one string into another" where an edit is an insertion, deletion or substitution.

For the strings $x[0..m)$ and $y[0..n)$, let $0 \le i \le m$ and $0 \le j \le n$, set $E[i, j]$ to be the edit distance between $x[0..i)$ and $y[0..j)$, and find $E[m, n]$.

There are three cases to be considered:

1. Cost= 1, to align $x[0..i-1)$ with $y[0..j)$ (insertion)
2. Cost= 1, to align $x[0..i)$ with $y[0..j-1)$ (deletion)
3. Cost= 1 if $x[i] \neq y[j]$ and 0 otherwise, to align $x[0..i-1)$ with $y[0..j-1)$.

By letting $\delta(i, j) := 1$ if $x[i] \neq y[j]$ and 0 otherwise, then

$$E[i,j] = \min\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\delta(i,j)\}$$

Algorithm 8: Levenshtein Distance

```
Input: Strings $x[0..m)$ and $y[0..n)$
Output: Edit distance between $x$ and $y$
1 for $i = 0$ to $m$ do $E[0, i] = i$;
2 for $j = 0$ to $n$ do $E[j, 0] = j$;
3 for $i = 1$ to $m$ do
4     for $j = 1$ to $n$ do
5         $E[i,j] = \min\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\delta(i,j)\}$
6 return $E[m, n]$;
```

1.2.5 Travelling Salesman Problem

For the complete undirected graph with vertex-set $\{0, 1, \ldots, n-1\}$ and edge lengths stored in the matrix $D = (d_{ij})$. Find a tour starting and ending at a specified node with minimum total length including all other vertices exactly once. This problem is *NP-hard*, as it is unlikely to ever be solved in polynomial time. A brute force technique of examining every path takes $O(n!)$ ($(n-1)!$ possibilities), but dynamic programming reduces this to $O(n^2 2^n)$.

By considering the subset $\{0, 1, \ldots, j\} \subseteq S \subseteq \{0, 1, \ldots, n-1\}$, let $C[S, j]$ but the shortest simple path length starting at 0 and ending at j, visiting each node in S exactly once. For $|S| > 1$, set $C[S, 0] = \infty$ (simple graph, therefore cannot start and end at same node). By expressing S in terms of its subproblems:

$$C[S, j] = \min\{C[S \setminus \{j\}, i] + d_{ij} \mid i \in S \wedge i \neq j\}$$

The required answer is therefore

$$\min\{C[\{O, 1, \ldots, n-1\}, j] + d_{j0} \mid 0 \le j < n\}$$

There are, at most $n \cdot 2^n$ subproblems, each taking linear time to solve, giving a total running time of $O\left(n^2 2^n\right)$.

1.2.6 All-pairs shortest path

Given a directed graph (V, E) with weight (considered as distance) $w : E \mapsto \mathbb{R}^{\ge 0}$, for each pair of vertices u and v, find the shortest path from u to v.

Suppose the vertex-set is $\{0, 1, \ldots, n-1\}$ and let $d[i, j; k] =$ length of shortest path from i to j, all of whose intermediate nodes are taken from $[0..k)$.

Initially,

$$d[i, j; 0] = \begin{cases} w(i, j) & \text{if } (i, j) \in E \\ \infty & \text{otherwise} \end{cases}$$

If we have $d[i, k; k]$ and $d[k, j; k]$ then the shortest path that from i to j that uses k, as well as other nodes, goes through k once, assuming no negative cycles. Hence, k is used in a shortest path from i to j iff $d[i, k; k] + d[k, j; k] < d[i, j; k]$, hence $d[i, j; k+1]$ should be updated accordingly. The running time is $O\left(|V|^3\right)$.

Algorithm 9: Floyd-Warshall Algorithm

```
Input: The directed graph $(V, E)$
       with weight (considered as distance) $w : E \mapsto \mathbb{R}^{\ge 0}$
Output: Shortest path between all pairs of nodes
1 for $i = 0$ to $|V| - 1$ do
2     for $j = 0$ to $|V| - 1$ do $d[i, j; 0] = \infty$;
3 for each edge $(i, j) \in E$ do $d[i, j; 0] = w(i, j)$;
4 for $k = 0$ to $|V| - 1$ do
5     for $i = 0$ to $|V| - 1$ do
6         for $j = 0$ to $|V| - 1$ do
7             $d[i, j; k+1] = \min\{d[i, k; k]+d[k, j; k], d[i, j; k]\}$;

8 return $d$;
```

1.3 Greedy Algorithms

Similar to dynamic programming algorithms, these are also used to solve optimisation problems, and work by only choosing the step with the most immediate benefit as the next step, without looking ahead, or reconsidering earlier decisions. They have the advantage that they are often more simple to implement, and there is no need for large amounts of storage, as only one decision is taken at each stage, and that decision is never reconsidered.

1.3.1 Change Making Algorithms

The Greedy Approach Start with no change, and at each stage, choose a coin of the largest denomination available that does not exceed the balance to be paid.

However, the above method does not work with all denominations of coins, and does not always yield the optimal solution. For example, with $100, 60, 50, 5, 1$, to pay 110, the greedy algorithm would give 3, $(100 + 5 + 5)$, whereas $60 + 50$ is a more optimal solution.

1.3.2 Minimum Spanning Tree Algorithm

Algorithm 10: Generic MST algorithm

```
1 $A = \emptyset$;
2 while $A$ is not a spanning tree do
3     find an edge $(u, v)$ that is safe for $A$;
4     $A = A \cup \{(u, v)\}$;
5 return $A$
```

Loop Invariant : "A is safe, i.e. a subset of some MST"
Initialisation : The invariant is trivially satisfied by $A = \emptyset$
Maintenance : Since only safe edges are added, A remains a subset of some MST.
Termination : All edges added to A are in an MST so A must be a spanning tree that is also minimal.

If $(S, V \setminus S)$ is a cut that respects A, and (u, v) is a light edge crossing the cut, then (u, v) is safe for A.

1.3.3 Kruskal's Algorithm

Description

- Start with each vertex being its own connected component.
- Find the edge with the lowest weight.
- Merge two components by choosing the light edge connecting them

Kruskal's requires a disjoint-set data structure to be most effective. It is a set of disjoint sets $\mathcal{S} = \{S_1, \dots, S_k\}$ where each set is represented by an individual element in each set.

Algorithm 11: Kruskal's Algorithm

```
  Input: The graph $(V, E)$ and weight function $w : E \mapsto \mathbb{R}$
  Output: An MST for the graph
1 $A = \emptyset$;
2 for each $v \in V$ do MAKE-SET($v$);
3 Sort $E$ by increasing weight $w$;
4 for each edge $(u, v)$ from the sorted list do
5     if FIND-SET($u$)$\neq$FIND-SET($v$) then
6         $A = A \cup \{(u, v)\}$;
7         UNION($u, v$);
8 return $A$;
```

Invariant Let $\mathcal{S}$ be the collection of sets in the disjoint-set data structure and L be the sorted list of edges not yet processed by the for-loop.

1. A is safe
2. For each $C \in \mathcal{S}$, $(C, A \restriction C)$ is a spanning tree of the subgraph $(C, E \restriction C)$.
3. Every processed edge's start- and end-points are in the same set in $\mathcal{S}$.

Initialisation $A = \emptyset$, $\mathcal{S}$ consists of only singleton sets and no edge has been processed $E \setminus L = \emptyset$ (hence all trivially true).

Maintenance Let $e = (u, v)$ be the edge to be processed, C_1 and C_2 be the sets that u and v belong to respectively, and A, $\mathcal{S}$ and L refer to the state before the iteration, and A', $\mathcal{S}'$ and L' be the state after the iteration.
If e is included in A, then C_1 and C_2 are different, and e is the minimum edge crossing the cut $(C_1, V \setminus C_1)$, and the cut respects A. As e is the next element to be processed, it must also be the lightest element, hence the cut lemma holds. The union operation ensures that the third part of the invariant holds.
If e is not to be included in A, then there are no changes to A, $\mathcal{S}$ and L is updated to include the discarded edge.

Termination All edges have been processed, therefore, $L = \emptyset$, and since all the nodes belong to the same set $C \in \mathcal{S}$, C spans the whole graph, and by application of the cut lemma, and safe edges is an MST.

Running Time Initialisation of A takes $O(1)$, first for-loop calls `MAKE-SET` $|V|$ times. E is sorted in $|E| \log |E|$ time. The second for-loop has $2|E|$ calls to `FIND-SET` and $|V| - 1$ calls to `UNION`, giving a total running time of $O(|E| \log |E|)$.

1.3.4 Prim's Algorithm

Description By growing the MST A from a given root node r, at each stage, find a light edge crossing the cut $(V_A, V \setminus V_A)$ where V_A is the edges incident on A.

The lightest edge can be found quickly by using a priority queue, where each entry in the queue is a vertex in $V \setminus V_A$. $key[v]$ is the minimum weight of any edge (u, v) where $v \in V_A$, the vertex returned by `EXTRACT-MIN` is v such that $\exists u \in V_A$ where (u, v) is a light edge crossing $(V_A, V \setminus V_A)$. $key[v] = \infty$ if v is not adjacent to any vertex in V_A.

Algorithm 12: Prim's Algorithm

```
  Input: The graph $(V, E)$ and weight function $w : E \mapsto \mathbb{R}$
  Output: An MST for the graph
1 $Q = \emptyset$;
2 for each $u \in V$ do
3     $key[u] = \infty$, $\pi[u] = \dagger$;
4     INSERT($Q, u$);
5 DECREASE-KEY($Q, r, 0$) while $Q \neq \emptyset$ do
6     $u$ = EXTRACT-MIN($Q$);
7     for each $v \in Adj[u]$ do
8         if $v \in Q \wedge w(u, v) < key[v]$ then
9             $\pi[v] = u$, DECREASE-KEY($Q, v, w(u, v)$);
```

Running Time Initialising Q takes $O(1)$, the first loop runs in $O(|V|)$, changing priority of r takes $O(\log |V|)$, and $|V|$ `EXTRACT-MIN` calls are required with, at most $|E|$ `DECREASE-KEY` operations, giving running time of $O(|E| \log |V|)$. The graph is connected, so $O(\log |E|) = O(\log |V|)$, hence total running time of $O(|E| \log |V|)$.

1.4 Dynamic Programming vs. Divide-and-Conquer

Dynamic programming is an optimisation technique, whereas divide-and-conquer is not normally used to solve optimality problems.

Both techniques split the input problems into smaller parts and use the solutions to the smaller parts to form a larger solution, however, dynamic programming solves the subproblems at all split points, whereas divide-and-conquer uses pre-determined split points using non-overlapping problems. Dynamic programming uses solutions to already calculated subproblems to find the total solution, to reduce space complexity.

2 Data Structures

2.1 Heaps

A heap is a type of tree without explicit pointers. Each level is filled from left to right, and the next level is only added when the previous is full. All heaps satisfy either the max-heap or min-heap property: "the value of a node (except the root node) is less than (greater than) or equal to that of its parent". In general, a heap can have any number of children on each of its nodes, and the maximum/minimum element of a max-/min-heap is at the root. Heaps are used as efficient priority queues, and for heapsort, which has a complexity of $O(n \log n)$.

2.1.1 Representation

The root is always at $A[0]$, and for any node $i > 0$, its parent is at $A[\lfloor (i-1)/2 \rfloor]$ and its left and right children are at $A[2i+1]$ and $A[2i+2]$.

2.1.2 Maintaining heaps

Algorithm 13: Heapify algorithm

```
   Input: Tree with left and right sub-trees of $i$ stored as heaps
   Output: $A$ where entire tree is also a heap.
1  $n = A.heapsize$;
2  $l = 2i + 1$, $r = 2i + 2$;
3  if $l < n \wedge A[l] > A[i]$ then $largest = l$ else $largest = i$;
4  if $r > n \wedge A[r] > A[largest]$ then $largest = r$;
5  if $largest \neq i$ then
6      exchange $A[i]$ with $A[largest]$;
7      HEAPIFY($A, largest$);
```

Running Time $\Theta(1)$ to find the largest of node and children. Worst-case has tree with last row half full (i.e. subtree rooted at i has, at most $2n/3$ elements), so $T(n) = T(2n/3) + \Theta(1) \Rightarrow T(n) = O(n^0 \log_{3/2} n) = O(\log n)$ by the Master Theorem.

Algorithm 14: Make-Heap algorithm

```
   Input: An unsorted integer array $A$ of size $n$
   Output: A heap of size $n$
1  $A.heapsize = A.length$;
2  for $i = \lfloor A.length/2 \rfloor$ to 0 do HEAPIFY($A, i$);
3  return $A$
```

Correctness

Invariant : each node $i+1, i+2, \ldots, n-1$ is the root of a heap for $-1 \leq i \leq \lfloor n/2 \rfloor$

Initialisation : each node $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n-1$ is a leaf, which is the root of a trivial heap, therefore the invariant holds.

Maintenance : calling HEAPIFY(A, i) causes i to become the root of a new heap, hence, when i is decremented, nodes at $i+1, i+2, \ldots, n-1$ are all roots of heaps.

Termination : when $i = -1$, the element at 0 is the root of a heap, therefore all elements below it are also roots of heaps.

Running Time There are n calls to `HEAPIFY`, each taking $O(\log n)$ time, giving $O(n \log n)$.

As `HEAPIFY` is linear with the height of the node that it runs on, the height of the heap is $\lfloor \lg n \rfloor$, hence the cost of `MAKE-HEAP` is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

As $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$, the total running time is therefore $O(n)$.