Incremental algorithms – Recursion 1) Find the subproblem – break the main problem down into smaller, easier to solve pieces. 2) Find the way to build one solution from another, or others. Two strategies for solving a problem incrementally: 1) solve next smaller problem, 2) use the subproblem solution to construct the solution for the full problem, OR 1) construct a small piece of the solution of the full problem, which simplifies the full problem, leaving a subproblem, 2) solve the subproblem. Example: insertion sort algorithm, where *sort()* procedure sorts the subarray A_N. The code for *Insert()* is not shown, it inserts A[N] into sorted elements via a loop that moves a given element left while it is less than the value to its left.

```
Require: 1 ⩽ N ⩽ A.length
 1: procedure SORT(A, N)
 2:     if N = 1 then
 3:         return
 4:     end if
 5:     SORT(A, N − 1)
 6:     INSERT(A, N)
 7: end procedure
```

```
 1: procedure INSERT(A, i)
 2:     tmp ← A[i]
 3:     j ← i
 4:     while j > 1 and A[j − 1] > tmp do
 5:         A[j] ← A[j − 1]
 6:         j ← j − 1
 7:     end while
 8:     A[j] ← tmp
 9: end procedure
```

```
 1: procedure SORT(A)
 2:     N ← A.length
 3:     for i ← 2 to N do
 4:         INSERT(A, i)
 5:     end for
 6: end procedure
```

Iterative version of InsertSort

Insert procedure: This algorithm *Insert()* moves the value that is initially at $A[i]$ to the left (towards position 1) until it is greater than or equal to the value on its left. The worst case occurs when $i=N$ and $A[N]$ is less than every value in $A[1,\dots,N-1]$. In this case $A[N]$ has to be compared with each of these values. So, the time complexity over all inputs has an upper bound of $O(N)$: $\boldsymbol{TInsert(N) = O(N)}$. The best case occurs when $A[i] \geq A[i-1]$, which requires just one comparison. The number of times the lines within the while loop execute is constant for all such inputs. So, the highest order term in the time complexity for this case is a constant, meaning there is a lower bound of $\Omega(1)$ on the time complexity for any input: $\boldsymbol{TInsert(N) = \Omega(1)}$. There is no function $f(N)$ for which $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$, so there is no Θ bound. **Recursive InsertSort procedure:** Worst case: array sorted in descending order. The num of iterations in calls of Insert: 1, 2, …, (N-1), and the total num of iterations (or comparisons) is $1 + 2 + \dots + (N-1) = \frac{N(N-1)}{2}$. In worst case, $T_{Sort}(N) = \Theta(N^2)$. Best case: array sorted in ascending order, so *Insert* will not perform any loops. For any input, we have: $\boldsymbol{T_{Sort}(N) = \Omega(N)}$ $\boldsymbol{T_{sort}(N) = O(N^2)}$. Analysing recursive algorithms: 1) write a formula directly from the code as a recurrence. 2) solve the recurrence. Based on InsertSort():

$$\begin{cases} T_{Sort}(N) = \Theta(1) & if\ N = 1 \\ T_{Sort}(N) = T_{Sort}(N-1) + T_{Insert}(N) + d & if\ N > 1 \end{cases}$$

Solving recurrence using substitution:

$T_{Sort}(N) = T_{Sort}(N-1) + (N-1)c + d'$ where we replaced $T_{Insert}(N)$.

Expand further by substituting the T_sort term on the RHS:

$T_{Sort}(N-1) = T_{Sort}(N-2) + (N-2)c + d'$ if $N > 2$ and replace:

$T_{Sort}(N) = T_{Sort}(N-2) + (N-2)c + (N-1)c + d' + d'$ if $N > 2$

Extract a general form:

$T_{Sort}(N) = T_{Sort}(N-i) + (N-i)c + \dots + (N-1)c + id'$ if $N > i > 0$. Which holds for $1 \leq i < N$. Setting $i = N-1$ will give formula where the only recursive term is the time for the base case:

$T_{Sort}(N) = T_{Sort}(1) + \big(1 + 2 + \dots + (N-1)\big)c + (N-1)d'$ if $N > i > 0$

We know that the time for base case is constant, so after solving the same arithmetic formula, we know that $\boldsymbol{T_{Sort}(N) = \Theta(N^2)}$. **Divide and conquer**: Principle: divide the main problem into multiple subproblems, break them down further, reapplying the same method, until they become trivially small. Can be applied in 2 ways: **1.** 1) Solve the next smaller set of problems, 2) use the subproblem solutions to construct the solution for the full problem. **2.** 1) Reconfigure the full problem, which must create a set of subproblems in such a way that solving the subproblems will complete the overall solution. 2) Solve the subproblem. **Divide and conquer sorting** – divide and redivide the array until trivially small, then sort as you merge, so that when merging, the subproblems are already sorted and all we need is to sort them altogether. **Mergesort and Merge**: All mergesort inputs are equivalent! – because the algo divides every input in the same way and does all the same merging. Formulas:

```
Require: 1 ⩽ L < R ⩽ A.length + 1
 1: procedure MERGESORT(A, L, R)
 2:     N ← R − L
 3:     if N = 1 then
 4:         return
 5:     end if
 6:     M ← L + ⌊N/2⌋
 7:     MERGESORT(A, L, M)
 8:     MERGESORT(A, M, R)
 9:     MERGE(A, L, M, R)
10: end procedure
```

```
Require: 1 ⩽ L < M < R ⩽ A.length + 1
 1: procedure MERGE(A, L, M, R)
 2:     left ← COPY(A, L, M)
 3:     right ← COPY(A, M, R)
 4:     i ← 1                                  // index in left
 5:     j ← 1                                  // index in right
 6:     for k ← L to R − 1 do
 7:         if i > left.length or (j ⩽ right.length and right[j] < left[i]) then
 8:             A[k] ← right[j]
 9:             j ← j + 1
10:         else
11:             A[k] ← left[i]
12:             i ← i + 1
13:         end if
14:     end for
15: end procedure
```

$$\begin{cases} T_{MSort}(N) = \Theta(1), & if\ N = 1 \\ T_{MSort}(N) = t_{MSort}\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T_{MSort}\left(\left\lceil \frac{N}{2} \right\rceil\right) + T_{Merge}(N) + d, & if\ N > 1 \end{cases}$$

$t_{MSort}(N) = 2T_{MSort}\left(\frac{N}{2}\right) + T_{Merge}(N) + d$ if $N = 2^j$ and $j \in \mathbb{N}$ and $j > 0$

Solve the recurrence (we know *Merge* takes $\Theta(N)$ time, so can replace it):

$T_{MSort}(N) = 2T_{MSort}\left(\frac{N}{2}\right) + Nc + d'$ if $N = 2^j$ and $j \in \mathbb{N}$ and $j > 0$.

Use a recursion tree to solve the recurrence:

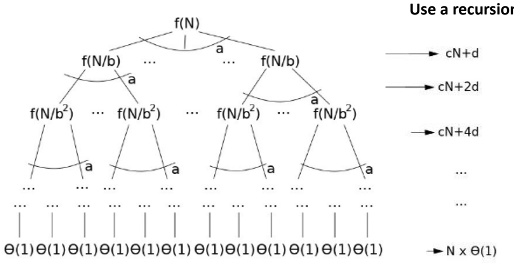


Figure 4.4: Recursion tree for a general divide and conquer algorithm.

$N \times \Theta(1)$ – time to solve the N base case problems. So, each level of the tree takes $\Theta(N)$ time. The height of the tree is log_2N, so $\boldsymbol{T_{MSort}(N) = \Theta(Nlog_2N)}$. **Master method** – used to solve recurrences in the form: $\boldsymbol{T(N) = aT\left(\frac{N}{b}\right) + f(N)}$ where $a \geq 1$ *and* $b > 1$ are constants. $f(N)$ is an asymptotically positive function, and $\frac{N}{b}$ can be replaced by either $\lfloor N/b \rfloor$ or $\lceil \frac{N}{b} \rceil$. Can be used to solve the formula from mergesort part, but with a=2, b=2, and $f(N) = cN + d'$. Divides the problem of size *N* into *a* subproblems, each subproblem is size N/b, and the time to divide the problem and combine the solutions is f(N). Asymptotic bounds on T(N) can be determined comparing $f(N)$ to $N^{\log_b a}$. There are **3 cases for the bounds on T(N)** – finding the highest order term: 1) The sum of the time for all the base cases could contain the highest order term, 2) f(N) and the base case sum could be of the same order. 3) f(N) could contain the highest order term. The heights of the recursion tree must be $\log_b N$. The tree has $\Theta(N^{\log_b a})$ leaves and this is the time taken for the base cases. The bounds on T(N) from the 3 cases: 1. If $f(N) = O(N^C)$, where $c < \log_b a$, then $T(N) = \Theta(N^{\log_b a})$. 2. If $f(N) = \Theta(N^{\log_b a} \times (\log_2 N)^k)$, then $T(N) = \Theta(N^{\log_b a} \times (\log_2 N)^{k+1})$. If $f(N) = \Omega(N^C)$, where $c > \log_b a$, then $T(N) = \Theta(f(N))$. **Quicksort algorithm** – Recursively sorts the array A_LR: 1) Divide A_{LR} into A_{LM} and A_{MR} so that sorting A_{LM} and A_{MR} will sort A. 2) Sort A_{LM} and A_{MR}. Divide A into two subarrays using the median value – *k – pivot value* so that A_LM contains only numbers that are smaller than any number in A_MR. Therefore, no need to sort after merging. The sorting is done within the subproblems. The problem of partitioning A can be solved in $\theta(N)$ time. 1. Partition $A_{L(R-1)}$ with pivot value $A[R-1]$. This defines the subarrays A_{LP} and $A_{P(R-1)}$. 2. Swap $A[R-1]$ with $A[P]$. Both numbered steps given above, partitioning $AL(R-1)$ and swapping $A[R-1]$ and $A[P]$ are assumed to happen within Partition. Time complexity of Quicksort: worst case inputs will cause the size of the partitions to decrease by a fixed amt in each recursive call, which causes InsertionSort-like behaviour, with $\Theta(N^2)$ time complexity. Inputs where the overall effect of partitioning is to reduce the size of the partitions by some fraction each time – incl. best cases – induce MergeSort-like behaviour, with $\Theta(Nlog_2N)$ time complexity. So: $T_{QSort}(N) = O(N^2)$ and $T_{QSort}(N) = \Omega(Nlog_2N)$. **Dynamic programming** - optimisation problem has many potential solutions that have to be searched through. All possible solutions should be checked. **The Rod Cutting Problem-** Given a rod of length N and a table P, where $P.length > N$, what is the maximum total value the company can get for the rod? (Example table P):A potential solution is a collection $\langle i1, \dots ik \rangle$ of positions in P such that $i_1 + \dots + i_k = N$. It can contain duplicates. The value of potential solution is $P[i_1] + \dots + P[i_k]$. For a rod with length N, there are 2^{N-1} ways to cut it up. We will use recursion, where subproblems are repeated a lot, and compare possible solutions by value. For each position $i > 0$ in the array P, there is a potential solution with value $r(i) = P[i] + R(N-1)$ (represents cutting off 1 piece of length I off the rod. Apply the formula for i=1 to N to capture all the possible places to to make the first cut and the max we could make in each case: $R(N) = \max_{i=1,N} r(i)$. Full definition:

```
Require: 1 ⩽ L ⩽ R ⩽ A.length + 1
 1: procedure QUICKSORT(A, L, R)
 2:     if R − L ⩽ 1 then
 3:         return
 4:     end if
 5:     P ← PARTITION(A, L, R)
 6:     QUICKSORT(A, L, P)
 7:     QUICKSORT(A, P + 1, R)
 8: end procedure
```

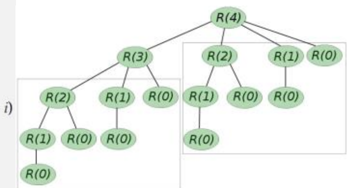
i	0	1	2	3	4	5	6	7	8	9
$P[i]$	0	2	5	7	6	11	14	17	18	21

$$\begin{cases} 0, & if\ N = 0 \\ \max_{i=1,N} P[i] + R(N-1), & if\ N > 0 \end{cases}$$

Where R(0) is the possibility of making no cuts.

Naive Recursive Rod Cut algorithm – correct but slow.

```
Require: N < P.length
 1: procedure RODCUTRECURSIVE(P, N)
 2:     if N = 0 then
 3:         return 0
 4:     end if
 5:     opt ← 0
 6:     for i ← 1 to N do
 7:         r ← P[i] + RODCUTRECURSIVE(P, N − i)
 8:         opt ← MAX(opt, r)
 9:     end for
10:     return opt
11: end procedure
```



Time complexity: $\boldsymbol{(2^N - 1)c + 2^{N-1}d + 2^{N-1}d' = \Theta(2^N)}$

$T(N) = \Theta(1)$, $\quad if\ N = 0$

$T(N) = T(N-1) + T(N-2) + \dots + T(0) + Nc + d$, $\quad if\ N > 0$

Solving the recurrence:

$T(N-1) = T(N-2) + \dots + T(0) + (N-1)c + d$, if N>1. Subtract:

$T(N) - T(N-1) = T(N-1) + c$, if N>1. Therefore:

$T(N) = 2T(N-1) + c$, if N>1. Replace N:

$T(N-1) = 2T(N-2) + c$ if N>2. Replace:

$T(N) = 4T(N-2) + 3c$ if N>2. Pattern:

$T(N) = 2^iT(N-i) + (2^i - 1)c$, if N>i. where $1 \leq i < N$

For $i = N-1$ we have: $T(N) = 2^{N-1}T(1) + (2^{N-1} - 1)c$.

Substituting: $T(1) = T(0) + c + d$ into the general formula:

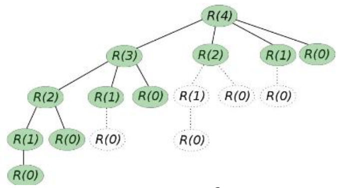
$T(N) = 2^{N-1}T(0) + 2^{N-1}c + (2^{N-1} - 1)c + 2^{N-1}d$. Therefore:

$T(N) = 2^Nc + 2^{N-1}T(0) + 2^{N-1}d - c = \Theta(2^N)$.

Top-down dynamic programming – solve each problem once and store the solutions in array R of length N+1.

```
Require: N < P.length
 1: procedure RODCUTTOPDOWN(P, N)
 2:     for i ← 0 to N do
 3:         R[i] ← 0
 4:     end for
 5:     return RODCUTAUX(P, R, N)
 6: end procedure

 7: procedure RODCUTAUX(P, R, N)
 8:     if N = 0 then
 9:         return 0
10:     end if
11:     if R[N] > 0 then
12:         return R[N]
13:     end if
14:     opt ← 0
15:     for i ← 1 to N do
16:         r ← P[i] + RODCUTAUX(P, R, N − i)
17:         opt ← MAX(opt, r)
18:     end for
19:     R[N] ← opt
20:     return R[N]
21: end procedure
```



Overall time complexity $\Theta(N^2)$ – solving N problems that each takes $\Theta(N)$. Because the full tree for R(N) has N internal nodes, but also N-1 more leaves than the tree for R(N-1). The number of leaves l(N): $l(N) = 1\ if\ N = 0$, and $l(N) = l(N-1) + N - 1$ if $N > 0$. So the total time within the leaves: $T_l(N) = \frac{N(N-1)}{2}d' + d' = \Theta(N^2)$. There is only 1 occurrence ofeach problem R(i) in the tree, for $1 \leq i \leq N$, total time for intern. nodes: $\boldsymbol{T_{in}(N) = (1 + \dots + N)c + Nd = \frac{N(N-1)}{2}c + Nd = \Theta(N^2)}$.

Recurrence: $\boldsymbol{T(N) = d, if\ N = 0. T(N) = T(N-1) + Nd, if\ N > 0.}$

Bottom-Up Dynamic Programming –

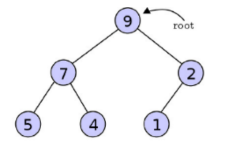
```
Require: N < P.length
 1: procedure RODCUTBOTTOMUP(P, N)
 2:     R[0] ← 0
 3:     for i ← 1 to N do                    // loop to find R[i]
 4:         opt ← 0
 5:         for j ← 1 to i do                 // try all pieces of length j ⩽ i
 6:             r ← P[j] + R[i − j]
 7:             opt ← MAX(opt, r)
 8:         end for
 9:         R[i] ← opt
10:     end for
11:     return R[N]
12: end procedure
```

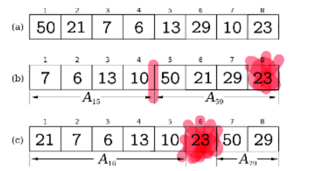
Arithmetic series & TC:

$c + 2c + \dots + Nc + d = \frac{N(N+1)}{2}c + d = \Theta(N^2)$

Greedy algorithm – used to find one solution to a problem, only acceptable when it is guaranteed to find an optimal solution. If we want to check if the solution is optimal, we have to attempt to disprove it.

The Fractional Knapsack Problem - What is the maximum value of gems that can be stolen if the thief's knapsack holds K grams? Theorem 5.1 (General Correctness of Greedy Choice). Let there be some (possibly empty) partial solution that is part of an optimal solution. Then there is an optimal solution that includes the partial solution plus the greedy choice. **Priority queues** – maintains a total order over its elements, each element has an associated value or key and there is a total order on the set of keys. Ex. FIFO queue. **Priority queue design** – each element has an explicit key. Max priority queue – next element removed is the one with top priority. Min priority queue – the opposite. **Binary heap design** – enqueue and dequeue run in $O(\log_2 N)$ time. In a max binary heap, each element's key is greater than or equal to the keys of all elements in its subheaps. Every element is the parent of maximum 2 children. **Adding an element to a max binary heap**. (a) The new element, with key 3, is added into the next free space. (b) The element





4.5: Designing a second divide and conquer sorting method (Quicksort). (a) The unsorted array. (b) … state would mean sorting A_{15} and A_{59} would complete the solution. (c) A "real" partitioning, … the pivot value 23.

is moved up towards the root until the correct ordering is achieved, by swapping with its parent(s). **Removing an element with the maximum key from a max binary heap**. (a) The root element is removed and replaced by the element from the last occupied position in the heap. (b–d) The new root element is moved down the heap until the correct ordering is achieved – compare with the keys of both its children and if necessary swap with the child with the greater key. The add/remove elements to/from the heap are localised to a single branch, so the time taken is $O(h)$, where h is the height of the heap. For a heap with N elements, $h = O(\log_2 N)$, so add/remove ops take $O(\log_2 N)$ time. **Implementation** – uses an array – root at index 0, then children, then their children. Use a stack-pointer-like var that keeps track of the first free space in the array. Parent, left ch, right child are at: $parent(i) = \lfloor\frac{i-1}{2}\rfloor$, $l_{child(i)} = 2i+1$, $r_{child(i)} = 2i+2$.

<u>**Dynamic array**</u> – when an element is added to the heap and the array isfull, create a new array with size 2N and copy the elements from the old array over to the new one.

```
Require: H.end > 0
procedure REMOVE(H)
    A ← H.array
    max ← A[0]
    H.end ← H.end − 1
    A[0] ← A[H.end]
    SINKROOT(A, 0, H.end)
    return max
end procedure

procedure SINKROOT(A, root, end)
    high ← root
    l ← LEFTCHILD(root)
    if l < end and A[l] > A[high] then
        high ← l
    end if
    r ← RIGHTCHILD(root)
    if r < end and A[r] > A[high] then
        high ← r
    end if
    if high ≠ root then
        SWAP(A, root, high)
        SINKROOT(A, high, end)
    end if
end procedure
```

```
procedure ADD(H, k)
    A ← H.array
    A[H.end] ← k
    H.end ← H.end + 1
    SWIMNODE(A, H.end − 1)
end procedure

procedure SWIMNODE(A, i)
    p ← PARENT(i)
    while i > 0 and A[i] > A[p] do
        SWAP(A, i, p)
        i ← p
        p ← PARENT(i)
    end while
end procedure
```

<u>**Amortised analysis**</u> – determining the time taken over a sequence of operations. For a dynamic array, an aggregate analysis shows that total time for a sequence of N add operations is $O(N\log_2 N)$, which is *amortised* $O(\log_2 N)$. **Method: Aggregate analysis**. The total time to add N elements is:

$T_{Total}(N) \le Nc + (1 + 2 + 4 + \cdots + (N-1))c$, if N>1, because the time to save a new element is constant (c), and the time to copy existing elements grows exponentially by the powers of 2. Even though the elements copied between the two arrays prior to adding the Nth elements could be less than N-1, we can still solve the geometric series:

$T_{total}(N) \le Nc + 2(N-1)c - c$, if N > 1.

$\le 3Nc - 3c$, if N > 1, → total amortised time.

$= O(N)$. Calculate the amortised time to add 1 element to an array containing N elements:

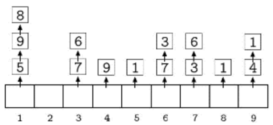
$T_{add}(N) \le \frac{3Nc-3c}{N}$, if N > 1;

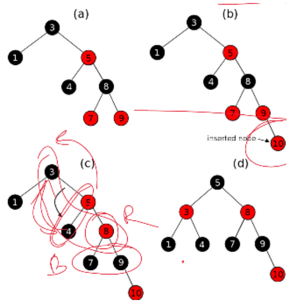
$\le 3c - \frac{3c}{N}$, if N > 1;

= amortised $O(1)$

The total time to add N elements to an empty heap is $O(N\log_2 N)$, so the time to add one element to the heap is amortised $O(\log_2 N)$. **Amortised Cost Methods.**

<u>**Dictionaries**</u>- set of (k, v) pairs. <u>**Binary search trees**</u> – pointer-based object, a tree is either the object NIL, or an object T with: (key of the root value, the data value stored at the root of the tree, left, and right (a BST). T.key is greater than all keys in T.left and less than all keys in T.right. (key, value) elements are always added to the tree as a new leaf; adding works exactly like search. Best case to add a new element is a tree T where T.key=inserted value k, and T.left=NIL, worst case is a single branch tree T with descending elements where k<k.leaf, and k.leaf is a leaf node of the tree T. $T_{search}(N) = O(N)\ and\ \Omega(1)$, $T_{add}(N) = O(N)\ and\ \Omega(1)$. <u>**Self-balancing trees**</u> – **red-black tree:** a RBT node is either NIL or has a colour





(R/B), key, value, parent, left, right. A RBT is a binary tree T where every element is a red-black tree node and for every non-NIL node n, n.key is > all keys in the binary tree with root n.left, and < all keys in the binary tree with root n.right. The root of T is Black, every leaf (NIL) in T is Black, children of Red are Black, all simple paths from each node to any descendant leaf contain the same num of Black nodes. Height $h = O(\log_2 N)$. The new leaf is always Red. Rebalancing is performed by rotation at node 3 and recoloring nodes 3 and 5. The RBT rebalancing is dense and not worth committing to memory – key features: proc maintains a ptr to a node n, initially the new leaf, as it proceeds, the proc executes a loop and the n ptr gets closer to the root in every iteration, every iter of the loop makes a const num of changes to the tree. At the end of the loop the numbered RBT properties are satisfied. $T_{add}(N) = O(\log_2 N)\ and\ T_{search}(N) = O(\log_2 N)$. <u>**Hash Table**</u> – array where each element is stored at an index determined by the key. *Trivial hashing* – stores element with key k in an array at index k. ***Collision*** – when 2 elements with diff keys are mapped to same position in the table. ***Hash function*** h – converts an obj k into a positive integer, used as the position at which to store an elem with key k

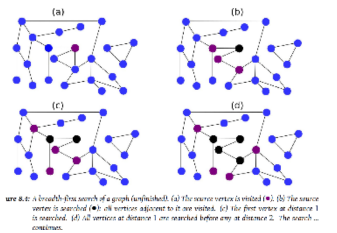
```
procedure INSERT(T, k, v)
    list ← T[h(k)]
    if there is an e in list where e.key = k then
        e.value ← v
    else
        create element x with x.key = k and x.value = v
        make x the head of list
    end if
end procedure
```

in a hash table, e.g. if key is numerical, h scales the value to be within 0 to m-1 range. Distinct keys should be mapped uniformly – similar keys mapped to diff parts of the table to minimise no of collisions. ***Chaining*** – places all the values mapped to the same position in the table into a linked list (see pseudocode). ***Simple uniform hashing*** – the probability of key k mapping to each position in the table is the same.

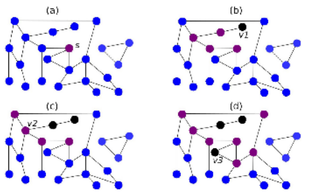
```
procedure INSERT(T, k, v)
    for i ← 0 to m − 1 do
        j ← h(k, i)
        if T[j] = NIL then
            create element x with x.key = k and x.value = v
            T[j] ← x
            return
        else if T[j].key = k then
            T[j].value ← v
            return
        end if
    end for
end procedure
```

Probing – used in open address hash tables, search and insert proc try diff positions in the table until a space is found (thru probe sequence). k_1 and k_2 that collide can't be stored at the same position. ***Uniform Hashing*** – Given a hash table T with m positions that uses hash fun h, a key k, h produces uniform hashing if the prob that $\langle h(k,0), \ldots, h(k, m-1)\rangle = \langle p_1, \ldots, p_{m-1}\rangle$ is the same for all permutations $\langle p_1, \ldots, p_{m-1}\rangle$ of $\langle 0, \ldots, m-1\rangle$. If $E[I] = I_1 \times P\{I = I_1\} + I_2 \times P\{I = I_2\}$, then $T(N) = \Theta(E[I])$, in general: $E[I] = \sum_{i=0}^{\infty} i \times P\{I = i\}$. Time complexity using chaining: $T(N) = \Omega(1)$ – time to compute h(k) + time to check the first element in the list T[h(k)]. Worst case – search for a key that isn't in the table - $T(N) = O(N)$. Average case (expected time): simple uniform hashing assumption (SUHA) – calc the expected num of times the search key k is compared to one of the keys in the table. Num of comparisons in table with m positions containing N keys is C. The prob of each one of keys having mapped to a position i is $1/m$. Expected length of chain: N/m. We search for key k that is not in the table. The prob that $h(k) = i$ is the same for each position in the table, each chain has the same expected length. $E[C] = N/m$. → *load factor*. So: $E[C] = \sum_{i=1}^{m} \frac{1}{m} \times \frac{N}{m} = \frac{N}{m}$. So expected time to search table with m positions, N keys, for key k under SUHA: $T(N,m) = \Theta(\frac{N}{m})$. If we use a resizeable array where the load factor can be kept below a const a: $T(N) = O(a) = O(1)$, and expected avg case time to insert a new element: $T(N) =$ *amortised* $O(1)$. Time complexity using probing: both for insert and search, $T(N) = \Omega(1)\ and\ T(N) = O(N)$. Average case: expected time under uniform hashing assumpt. From $E[C] = \sum_{i=1}^{\infty} P\{C = i\}$, we could limit it at m: $E[C] = \sum_{i=1}^{\infty} P\{C \ge i\}$ – the prob of at least i comparisons occurring for all $i \ge 1$. For $i = 1$, the prob is 1. $P\{C \ge i\} =$ $P\{first\ i\ positions\ are\ occupied\}$, if $i \ge 2$. There are N occupied positions out of m, so the prob of trying an occupied position first is N/m, so: $P\{C \ge 2\} = P\{first\ 1\ position\ is\ occupied\} = \frac{N}{m}$. If first position was occupied, it leaves m-1 other positions to try, of which N-1 are occupied: $P\{C \ge 3\} = P\{first\ 2\ positions\ are\ occupied\} = \frac{N}{m} \times \frac{N-1}{m-1}$. Generally: $P\{C \ge i\} \le \left(\frac{N}{m}\right)^{i-1}, if\ i \ge 1$. Since $P\{C \ge i\} = 1$, then: $E[C] \le \sum_{i=1}^{\infty} \left(\frac{N}{m}\right)^{i-1} \to \le \sum_{i=1}^{\infty} \left(\frac{N}{m}\right)^{i}$, which is a geometric series, so: $E[C] \le \frac{1}{1-\frac{N}{m}} = \frac{m}{m-N}$. It is often shown as: $E[C] \le \frac{1}{1-\alpha}$ where $\alpha = N/m$. When $N = 0$ the expected num of probes is 1, to the first position in the probe sequence. As $N \to m$ the expected num of probles tends to ∞ (or m). If N/m is limited to a constant by resizing as more elements are added, then the expected time search: $T(N) = O(1)$ and for insert $T(N) = amortised\ O(1)$.

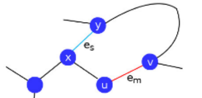
<u>**Graphs**</u> – A Graph G =(V, E) is a set V of objects – vertices, and a set E of pairs of vertices {u,v} -edges. If there is an edge {u,v} in E, then vertex u is adjacent to vertex v, and v is adjacent to u. An edge



{u,v} is also written as uv. Undirected – without a connection direction. Adjacency list – array A of length $|V|$, where $A[i]$ is a linked list containing the vertices adjacent to vertex i. Adjacency matrix- $|V| \times |V|$ matrix M, e.g. 2-d array. Searching/traversing a graph – graphs have no beginning/ end and are all equivalent, select vertex s – source. Each vertex is visited and searched. **Breadth-First Search** – ordered according to the distance of each vertex from the source, as measured in edges crossed. **Shortest Paths** – BFS algorithm assumes that the vertices of G are natural nums 1 to $|V|$. g is an adjacency list representation of G, so g is an array of len $|V|$, of linked lists of vertices, BFS is implemented using a FIFO queue, initialised to contain s. **Depth-First Search** – rapidly increases the distance from source by following a single path through the graph as far as possible. The search backtracks to the most recent alternative path, follow that, and so on. DFS uses a stack. DFS-Component is called for all unvisited vertices adjacent to u. **Time complexity** – For a graph G=(V,E) with $|V|$ vertices and $|E|$ edges, BFS runs in $O(|V| + |E|) = T_{BFS}(V, E) = O(V + E)$ time and



DFS: $T_{DFS}(V, E) = \Theta(|V| + |E|)$. **Weighted undirected graph** – each edge has a weight/cost $\{u, v\} \in E$, written $w(u, v)$. **Spanning Tree** – given a graph G=(V,E), a spanning tree of G is a tree $T = (V, E_T)$ such that $E_T \subseteq E$. A way of connecting all the vertices of a graph using a subset of edges, without any cycles. **Minimum spanning tree** of a connected, weighted graph G is a spanning tree $T(V, E_T)$ of G such that the total weight of T: $\sum_{\{u,v\} \in E_T} w(u, v)$ is less or equal to the total weight of all other spanning trees of G. Finding a MST is an optimisation prob. T will include all the vertices of G with some edges, should be a minimum weight set of $|V| - 1$ edges that don't form a cycle. **MST Greedy choice** – if e_m is the minimum weight edge in the set $E - E_T$ such that the graph $(V, E_t \cup \{e_m\})$ is acyclic, then $E_t \cup \{e_m\}$ is a subset of the edges of a minimum spanning tree of G. **Kruskal's Algorithm** – builds an MST for a weighted connected graph $G = (V, E)$ by iterating through edges in E in ascending order by weight. Start with all vertices unconnected, so each vertex is separate, as edges are added, the vertices start to



become connected. When edge is considered for inclusion in T, algo has to check whether u and v are in the same component – if yes, it cannot be added bc of cycle. Check belonging to components using **Union-Find data structure/ Disjoint Set Forest** – manages a collection of elements that are organised into disjoint sets. Operations: Union(x,y) – unifies sets containing x,y; and Find(x) – find set cont elem x. Time complexity for both union and find is $O(N)$. Performing N-1 unions, which must reduce the forest to a single tree assuming always unifying distinct sets, can create a tree with 1 branch, length N. Finding set containing elem at the end of the branch – N iterations of the loop in the Find proc. **Union by rank** – each object has a rank – the height of the subtree with that obj as a root. The shorter tree becomes subtree of the root of the longer one. The height of a tree only increases when 2 trees of equal height are unified. **Path compression** – (within Find(x)) – when finding the root of the tree containing element x, we set the parent of every node on the path between x and the root to be root. Union-find with union-by-rank and path compression time complexity: $T(N, M) = O(M\ \alpha(N))$ where α is a very slow growing function. Time complexity of Kruskal's algo: depends

```
procedure MAKESET(x)
    x.parent ← x
end procedure

procedure FIND(x)
    node ← x
    while node.parent ≠ node do
        node ← node.parent
    end while
    return node
end procedure

procedure UNION(x, y)
    xroot ← FIND(x)
    yroot ← FIND(y)
    xroot.parent = yroot
end procedure
```

on the priority queue and union-find data structures. Total time: $T(E, V) = \Theta(V) + O(E\log_2 E) + O(E\log_2 V) + O(V\log_2 V)$ Since the graph input is connected, $E \ge V - 1$, and so $V = O(E)$, which simplifies: $T(E, V) = O(E\log_2 E) + O(E\log_2 V)$. Since $E < V^2$ in any graph, $\log_2 E = O(\log_2 V)$, so: $T(E, V) = O(E\log_2 V)$.

```
procedure MAKESET(x)
    x.parent ← x
    x.rank ← 0
end procedure

procedure FIND(x)
    if x.parent ≠ x then
        x.parent ← FIND(x.parent)
    end if
    return x.parent
end procedure

procedure UNION(x, y)
    xroot ← FIND(x)
    yroot ← FIND(y)
    if xroot.rank < yroot.rank then
        xroot.parent ← yroot          // no increase in heig
    else
        yroot.parent ← xroot
        if xroot.rank = yroot.rank then       // height increases
            xroot.rank ← xroot.rank + 1
        end if
    end if
end procedure
```

```
procedure KRUSKAL(g)
    T = [g.length]                    // Tree being constructed
    q ← EMPTYQUEUE()                  // min priority queue
    for u ← 1 to g.length do
        nodes[u] ← UFNODE()
        MAKESET(nodes[u])             // add vertex to union-find
        for v in g[u] do
            if u < v then             // queue each edge once
                ENQUEUE(q, (u, v), w(u, v))   // weight is priority
            end if
        end for
    end for
    while not EMPTY(q) do
        (u, v) ← DEQUEUE(q)
        if FIND(nodes[u]) ≠ FIND(nodes[v]) then
            ADDEDGE(T, u, v)
            UNION(nodes[u], nodes[v])
        end if
    end while
    return T
end procedure
```

```
Require: comp.length = g.length
Require: g.length > 0
procedure COMPONENTS(g, comp)
    for i ← 1 to g.length do
        comp[i] ← −1                  // component number of each vertex
    end for
    c ← 0
    for v ← 1 to g.length do
        if comp[v] = −1 then          // vertex not in searched component
            c ← c + 1
            DFS-COMPONENT(g, v, c, comp)
        end if
    end for
    return c
end procedure

Require: comp[u] = −1
procedure DFS-COMPONENT(g, u, c, comp)
    comp[u] ← c
    for v in g[u] do
        if comp[v] = −1 then          // previously unvisited
            DFS-COMPONENT(g, v, c, comp)
        end if
    end for
end procedure
```

```
Require: dist.length = g.length
Require: parent.length = g.length
Require: 1 ≤ s ≤ g.length
procedure SHORTEST-PATHS(g, s, dist, parent)
    for i ← 1 to g.length do
        dist[i] ← ∞
        parent[i] ← −1
    end for
    q ← EMPTYQUEUE()                  // FIFO queue
    ENQUEUE(q, s)
    dist[s] ← 0
    while not EMPTY(q) do
        u ← DEQUEUE(q)
        for v in g[u] do
            if dist[v] = ∞ then       // previously unvisited
                dist[v] ← dist[u] + 1
                parent[v] ← u
                ENQUEUE(q,v)
            end if
        end for
    end while
end procedure
```