COMP 6651
Algorithm Design Techniques

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Table of Contents

Greedy Algorithms (§15.1-15.3)

The greedy paradigm is used in many optimization algorithms: a solution is sought that optimizes (minimizes or maximizes) an objective function.

It is called *greedy* because when a choice is to be made, it chooses what looks best at that moment. A greedy strategy usually progresses in a top-down fashion, committing to a choice, then solving sub-problems of the same type.

Many problems have the greedy-choice property: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

We will study examples where the greedy approach

1. gives optimal solutions;
2. good approximation–gives a simple approximation algorithm where an exact solution is too difficult to get;
3. is not suitable (e.g., give a poor approximation).

Optimal Binary Search Trees

Consider the following probabilities for nodes in a binary search tree (BST).

node	*a*	*b*	*c*	*d*
p_i	0.35	0.15	0.25	0.2

Recall that $E[T]$ is the cost of a tree T expresses the expected number of comparisons corresponding to the given probabilities of searches.

The goal is to find an optimal BST T, a BST that gives lowest expected cost.

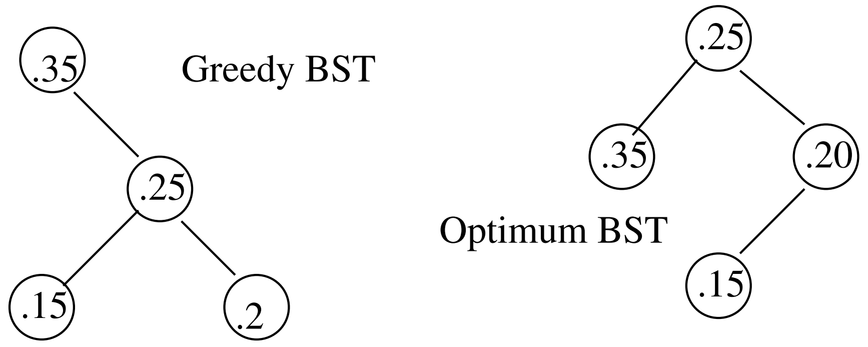
Example of a greedy BST algorithm:

Make the key with the largest probability of search the root r of BST.

The choice of r splits the keys into two sublists, each sublist will be a subtree of r.

Do the same procedure for each subtree of r.

node	a	b	c	d
p_i	0.35	0.15	0.25	0.2



So in the case of constructing an optimal BST, the greedy approach does not give an optimal solution.

Actually, we can give an example where the greedy BST is very bad:

node	a_1	a_2	a_3	...	a_n
p_i	$\frac{1}{n} - \frac{1}{n^3}$	$\frac{1}{n} - \frac{2}{n^3}$	$\frac{1}{n} - \frac{3}{n^3}$	...	$\frac{1}{n} - \frac{1}{n^2}$

Since the probabilities are decreasing linearly (as selected by the greedy algorithm), if $a_1 < a_2 < a_3 < \cdots < a_n$, greedy BST creates a singly linked list, thus the cost is at least

$$\sum_{i=1}^{n} i(1/n - i/n^3) \geq \sum_{i=1}^{n} i(1/n - 1/n^2) > n/4$$

However a balanced BST gives a cost at most $\log n$. In this case, it does not give even a good approximation of an optimal solution.

However, the greedy approach works for many optimization problems.

Thus, when we use the greedy paradigm, we must verify that it is appropriate for the given problem.

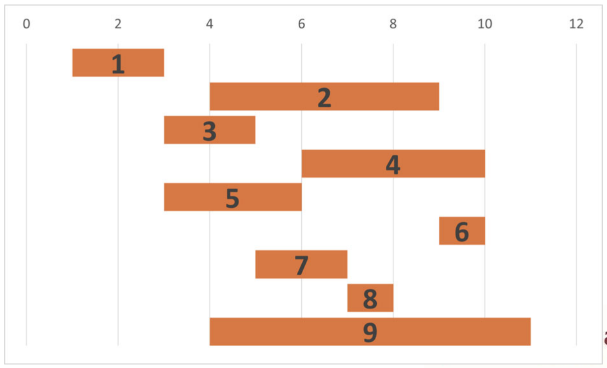
Activity Selection Problem (§15.1)

We have a set of n proposed activities, $S = \{a_1, a_2, \ldots, a_n\}$, for **a single lecture hall**. Each activity a_i has a start time s_i and finish time f_i. Find the *largest* set of activities that can be held in the lecture hall that respect the start - finish times.

i	1	2	3	4	5	6	7	8	9
s_i	1	4	3	6	3	9	5	7	4
f_i	3	9	5	10	6	10	7	8	11

Two requests are *compatible* if their time slots do not overlap.

requests 1 and 5 are compatible,
requests 2 and 5 are not compatible.



What is the greedy choice for the scheduling problem if we want to schedule as many activities as possible:

Schedule as first e_1, the activity a_i that terminates first.

For the next event take the activity that starts after the end of the previous event and terminates as early as possible, etc.

Is this strategy optimal?

Take a schedule $e'_1, e'_2, \ldots, e'_k$ that is an optimal schedule.

If $e'_1 \neq e_1$, then we can replace e'_1 with e_1 since e'_1 does not terminate earlier than e_1 and get an optimal schedule.

The same can be said then about the second activity, etc.

Thus, the greedy strategy in the case produces an optimal schedule.

i.e., a local optimization produces a globally optimal solution.

Possible natural orders

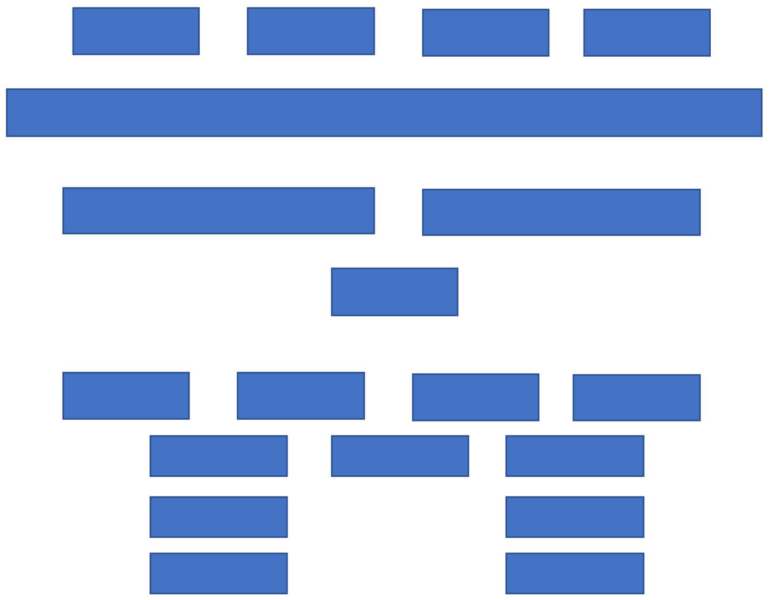
For our optimal solution, we ordered the activities by **earliest finish time**, i.e., consider jobs according to non-decreasing f_i.

Other possibilities are:

- **Earliest start time**: consider jobs according to non-decreasing s_i
- **Shortest interval**: consider jobs according to non-decreasing $f_i - s_i$
- **Fewest conflicts**: for each job i, count the remaining number of conflicting jobs c_i. Schedule according to non-decreasing c_i

None of these other possible job orderings also leads to an optimal greedy algorithm for this problem, as we see by the following counterexamples.

Counterexamples


Earliest start time counterexample
Shortest interval counterexample
Fewest conflicts counterexample

We need to search the list of activities for the one that starts after the end of the last event so far with earliest termination.

If requests are not sorted, we need to find up to $n-1$ times the *minimum* termination time among i activities, $1 \leq i \leq n$.

This gives $O(n^2)$ algorithm.

Better way
pre-sort all activities by termination time in monotonically increasing order such that $f_i \leq f_{i+1}$:
needs $O(n \log n)$ time.

To find up to $n-1$ times the *minimum* termination time among i sorted activities, $1 \leq i \leq 2$ is done in $O(n)$ time.

$O(n \log n) + O(n) = O(n \log n)$ time.

Greedy algorithm:

```
GREEDY-ACTIVITY-SELECTOR($s, f, n$)
  \\ $s$,$f$ are lists of start,finish times, sorted by monotonically increasing finish time
  \\ $n$ is number of activities
  $A \leftarrow \{a_1\}$
  $k \leftarrow 1$
  for $m \leftarrow 2, n$ do
      if $s[m] \geq f[k]$ then                    ▷ is $a_m$ in $S_k$?
          $A \leftarrow A \cup \{a_m\}$                ▷ yes, so choose it
          $k \leftarrow m$                      ▷ and continue from there
```

In the above, $S_k = \{a_i \in S : s_i \geq f_k\}$ is the set of activities that start after activity a_k finishes.

Example

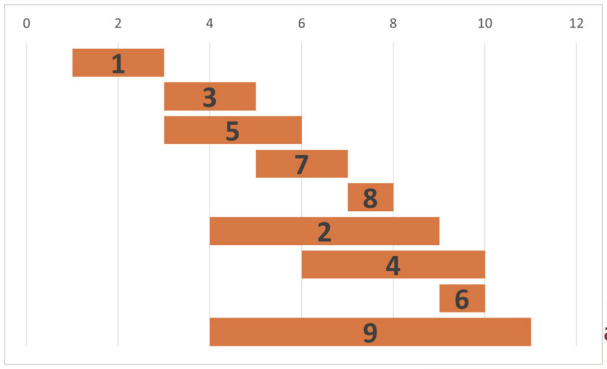
i	1	2	3	4	5	6	7	8	9
s_i	1	4	3	6	3	9	5	7	4
f_i	3	9	5	10	6	10	7	8	11

Sort by terminations:

i	1	3	5	7	8	2	4	6	9
s_i	1	3	3	5	7	4	6	9	4
f_i	3	5	6	7	8	9	10	10	11

Optimal Schedule:

#1, #3, #7, #8, #6
i.e., 5 activities can be held in the hall.



Scheduling Problems

We have activities to schedule as before,
each activity has a start time and finish time.

Find the ***smallest* number of rooms** needed to schedule *all* activities.

Main difference is that all activities must be scheduled.

In what order we would do it in the greedy manner?

Schedule an activity in a room already used in scheduling whenever possible.

Only when next activity request conflicts with activities in all rooms in use, increase the number of rooms by one.

Consider the requests in the *order of their start time*.

List of activities $a_1, a_2, \ldots a_n$ sorted by start time.

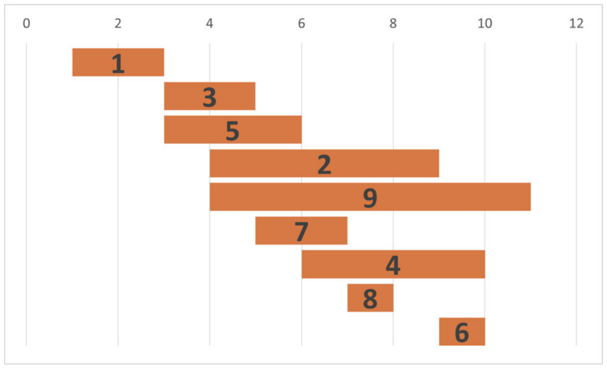
Schedule request a_1 in room 1.

If a_2 starts after the finish of a_1,
put it in room 1,
else in room 2, etc.

$\vdots$

If an activity a_i can be put in one of the rooms already in use, put it there, else increase the number of rooms and schedule a_i in that new room.

This produces a schedule.



Is the greedy scheduling strategy optimal in this case?

If we increase the number of rooms from m to $m + 1$ when scheduling activity a_i, we do it because all rooms 1 to m contain events whose

starting times are $\leq s_i$ and finish times are $> f_i$.

Thus there are $m + 1$ requests that are incompatible with one another.

Conclusion:
We need at least $m + 1$ rooms.

This implies that the strategy produces an optimal schedule.

```
GREEDY-ACTIVITY-SCHEDULE($s, f, n$)
    \\ $s$,$f$ are lists of start,finish times, sorted by monotonically increasing start time
    \\ $n$ is number of activities
    $m, j \leftarrow 0$                                    ▷ $m$ is the number of rooms needed
    $l[1..n] \leftarrow -1$                                ▷ assuming $f[i] \geq 0$
    \\ $l[k]$ is the finish time of the last activity in room $k$ so far
    for $i \leftarrow 1, n$ do
        for $j \leftarrow 1$; $((j \leq m))$ & $(l[j] > s[i]))$; $j \leftarrow j + 1$ do
            \\ search for available room
        if $((j > 0)$ & $(j \leq m))$ then
            $l[j] \leftarrow f[i]$; print "$i : j$"         ▷ put activity $a_i$ in room $j$
        else
            $m \leftarrow m + 1$                           ▷ one more room is needed
            $l[m] \leftarrow f[i]$; print "$i : m$"
    return $m$
```

What is the run-time of the algorithm?

There are two nested loops.

Outer loop is repeated n times,
Inner loop is repeated up to $i - 1$ times:

Run-time is

$$O(\sum_{i=1}^{n} i) = O(n(n+1)/2) = O(n^2)$$

Can we make it $= O(n \log n)$?

The problem is that we use a *linear search* to find out if there is a room that can accommodate the next activity.

The next activity can be put into any room that is available at the start time of the activity.

So we can as well put the activity in the room that is available earliest, if compatible.

Use a priority queue to store information about the end of the last event in each room.

Implement the priority queue as a heap. The top element in the heap is the room with earliest termination time (i.e., a min-heap).

In a heap, inserting a new element and removing the minimal element are $O(\log n)$ operations.

```
**GREEDY-ACTIVITY-FAST-SCHEDULE**$(s, f, n)$
  \\ s,f are lists of start,finish times, sorted by monotonically increasing start time
  \\ n is number of activities
  $m \leftarrow 0$                                   ▷ initially, we have 0 rooms
  $Q \leftarrow \emptyset$                           ▷ Initialize min-priority queue, Q
  Q.insert$(m, max\_int)$
  for $i \leftarrow 1, n$ do
      Q.removeMin$(room, last)$           ▷ room is the room number; last is $f[room]$
      if $last > s[i]$ then
          Q.insert$(room, last)$ ▷ finish time is after start time for $a_i$; can't use room
          $m \leftarrow m + 1$                       ▷ one more room needed
          print "$i : m$"
          Q.insert$(m, f[i])$
      else
          print "$i : room$"
          Q.insert$(room, f[i])$
  return $m$
```

$\boxed{\text{run-time} = O(n \log n)}$

Notice that the two scheduling problems use different greedy choices:

1. One uses the starting time,
2. the other the finishing time of the activity

There are many type of scheduling problems, in spite of similarities they can be very different in their complexity.

General Scheduling Problem

We are given a set of activities for scheduling.

For each activity, we indicate:

1. the duration of activity,
2. which other activities the event activity is incompatible with

Common to class-room scheduling since some courses cannot be scheduled at the same time if the same group of students is supposed to take them.

Goal:
Find a schedule that minimizes the number of rooms and respects the incompatibilities.

Here a greedy strategy does not work.

Why the difference?

We don't have any fixed start, finish times that are used in the scheduling.

We will see later on that this problem is computationally very difficult.

All known algorithms are exponential,
we will show the problem is *NP*-complete (last part of the course).

Knapsack problems (§15.2)

We have a container of size W and a set of n items $i_1, i_2, \ldots, i_n$.

Each item i_j has weight w_j and value v_j.

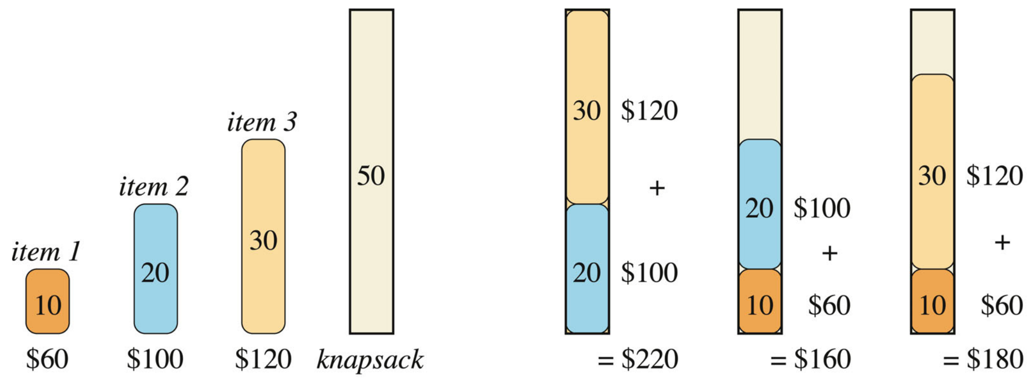
0-1 Knapsack Problem:
Find a subset of items which together has weight $\leq W$ and that maximizes the total value.

This problem *cannot* be solved by a greedy algorithm.

Greedy choice: take items in order of their value per unit of weight:

Example:
$W = 50$, $\{(w_1 = 10, v_1 = \$60), (20, \$100), (30, \$120)\}$

Value per unit weight $= (\$6, \$5, \$4)$



The optimal subset includes items 2 and 3. Any solution with item 1 (with greatest value per unit of weight) is suboptimal.

It is also an *NP-complete* problem in general (will be shown later).

0-1 Knapsack Problem: Dynamic Programming Solution

We can solve this problem in time $O(n \cdot W)$ with dynamic programming

This is called pseudo-polynomial time (more on this later)

Determining sub-problems:

- We can reduce the problem size by reducing capacity W
- We can reduce the problem size by restricting to the first j inputs $[w_1, \ldots, w_j]$ and $[v_1, \ldots, v_j]$
- Turns out, we need both

Assume w_k, v_k, and $W \in \mathbb{N}$

Consider items $[w_1, \dots, w_j]$, $[v_1, \dots, v_j]$ and capacity W

Optimal solution *OPT* for this instance either:

- Doesn't include item i_j: it must be an optimal solution to

$$[w_1, \dots, w_{j-1}], [v_1, \dots, v_{j-1}] \text{ and capacity } W$$

- Includes item i_j: then it collects value v_j and must contain an optimal solution to

$$[w_1, \dots, w_{j-1}], [v_1, \dots, v_{j-1}] \text{ and capacity } W - w_j$$

Note that item i_j is included only if $w_j \leq W$

Proof: argument by induction

Computing optimal value

```
Iterative_DP_0-1_Knapsack($w[1..n], v[1..n], W$)
  Instantiate $D[0..n, 0..W] \leftarrow 0$
  for $j \leftarrow 0, n$ do
      $D[j, 0] \leftarrow 0$
  for $c \leftarrow 0, W$ do
      $D[0, c] \leftarrow 0$
  for $j \leftarrow 1, n$ do                                        ▷ $O(n)$
      for $c \leftarrow 1, W$ do                                    ▷ $O(W)$
          $D[j, c] \leftarrow D[j-1, c]$
          if $w[j] \leq c$ then
              $D[j, c] \leftarrow \max(D[j, c], D[j-1, c-w[j]] + v[j])$
  return $D[n, W]$
```

Overall running time is $O(n \cdot W)$

Why is running time $O(n \cdot W)$ not polynomial?

Observe that to write down value W as part of the input we only need $\log W$ bits

Therefore, input length is expressed in terms of n and $\log W$

Polynomial running time refers to polynomial in the input length

Therefore, polynomial running time for Knapsack would be expressed as a polynomial in terms of n and $\log W$

Our running time is $n \cdot 2^{bitlength(W)}$, which is exponential in input length

Running times which are polynomial in numerical values are called pseudo-polynomial

Example

Suppose that $n = 10000$, $w[1..10000]$, $v[1..10000]$ and

$$W = 999999999999$$

Observe that this value of W requires only 12 digits to specify. So it is a very short input, although it stands for a very large number

Our algorithm would run in time proportional to $n \cdot W$=9999999999990000

That's a long time to wait!

Fractional Knapsack Problem

Find a subset of items **or fractions of items** which together have weight $\leq W$ and that maximizes the total value.

Since we can take fractions, we can always fill-up the knapsack to maximum.

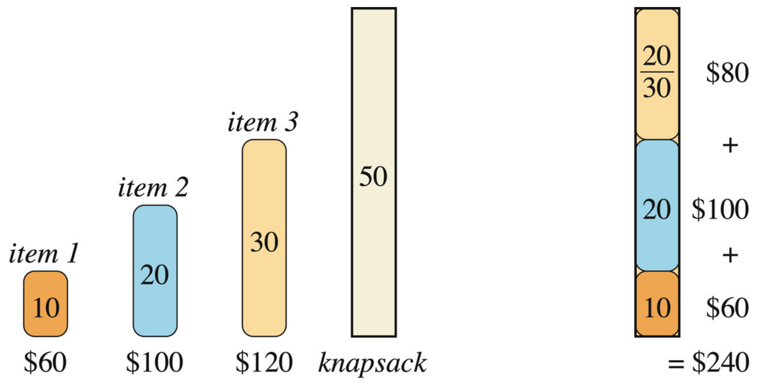
$W = 50$, $\{(w_1 = 10, v_1 = \$60), (20, \$100), (30, \$120)\}$

Take items or a fraction of an item in order of value per unit of weight until knapsack is full.

Divisibility property of items makes the fractional knapsack problem much easier!

Value per unit weight $= (\$6, \$5, \$4)$

Optimum: $\{i_1, i_2, \frac{20}{30} i_3\}$



The same example as before showing that the greedy strategy works for the fractional knapsack problem.

For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution. Any optimal solution must include the most expensive (per unit of weight) items, otherwise we can replace part of the content of a knapsack with larger value.

In the fractional knapsack problem, we can replace any part of the knapsack content with items having the same volume.

In $0-1$ knapsack content this argument cannot be used. If we try to replace some item(s), in the knapsack:

- The replacement does not need to fit in (and we cannot take a fraction) or it leaves some empty space, which is not necessarily optimal globally.

This is the main reason why a local decision in $0-1$ knapsack is not necessarily optimal and why we cannot use a greedy algorithm to get an optimal solution.

Natural greedy algorithm

Indeed, since items are divisible it makes sense to start taking as much of an item that gives the biggest bang for the buck as possible

Sort items in non-increasing order of v_i/w_i (value per unit weight)

Process items in this order and for each item:

if the remaining capacity of the knapsack can accommodate the entire item, take the entire item
otherwise take the largest portion of the item you can filling the knapsack to capacity

Computing optimal value

```
Fractional_Knapsack($w[1..n], v[1..n], W$)
    Instantiate and initialize $X[1..n] \leftarrow 0$    ▷ to keep track of the original unsorted order
    Instantiate and initialize $I[1..n] \leftarrow \{1, 2, \ldots, n\}$    ▷ to keep track of the original indices
    sort $w[], v[], I[]$ simultaneously in non-increasing order of $v_i / w_i$
    for $j \leftarrow 1, n$ do
        $X[I[j]] \leftarrow \min(W, w[j]) / w[j]$
        $W \leftarrow W - X[I[j]] \cdot w[j]$
    return $X$
```

Overall running time is dominated by the sorting procedure, so $O(n \log n)$

Huffman codes (§15.3)

They are used for compression of text.

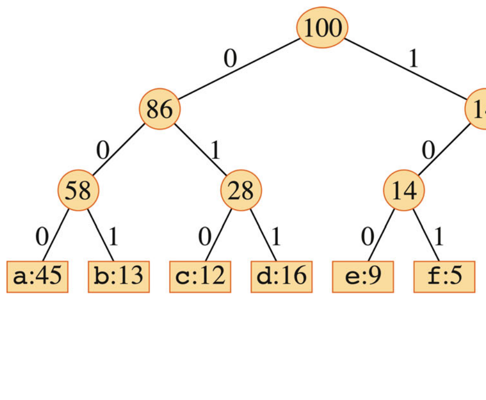
Instead of representing each character by a **fixed-length code** (7 or 8 bits typically), most frequently used characters are given very short (binary character) code while less frequently used characters are given longer codes... **variable-length code**

To allow easy decoding of files written in variable length codes, we use **prefix-free code** in which no code is a prefix of another code (otherwise we would not know when a character ends).

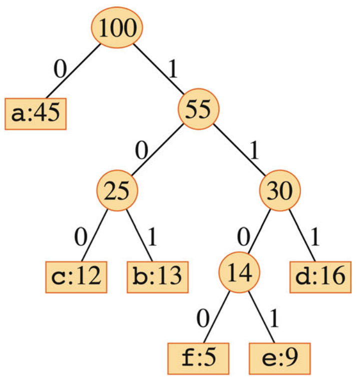
$100, 1001$ are not prefix-free codes.

$0, 10, 110, 1110, 11110$ is prefix-free code.

Fixed Length versus Prefix-Free coding



(a)



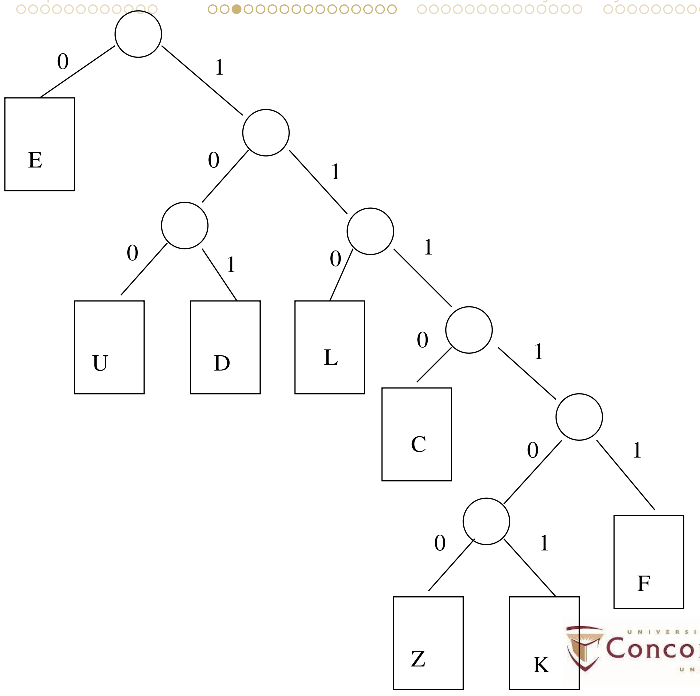
(b)

(a) The tree corresponding to the fixed-length code a = 000, b = 001, c = 010, d = 011, e = 100, f = 101. (b) The tree corresponding to the optimal prefix-free code a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100.

Prefix-free codes can be obtained from full binary trees in which the characters are leaves.

Code $=$ path to the character (also called the codeword for the character).

E = 0, U = 100, D = 101, L= 110, F=11111 etc.



Problem: Build a coding tree T that optimizes the compression based on the known frequency of characters:

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$.

Let $d_T(c)$ denote the depth of c's leaf in the tree T. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode an alphabet is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c),$$

which we define as the cost of the tree T.

Building a Huffman prefix-free coding tree:

1. order the characters by frequencies ($c.freq$) starting from smallest to largest.

repeat the following:

2. Join first two elements of the list into a tree.

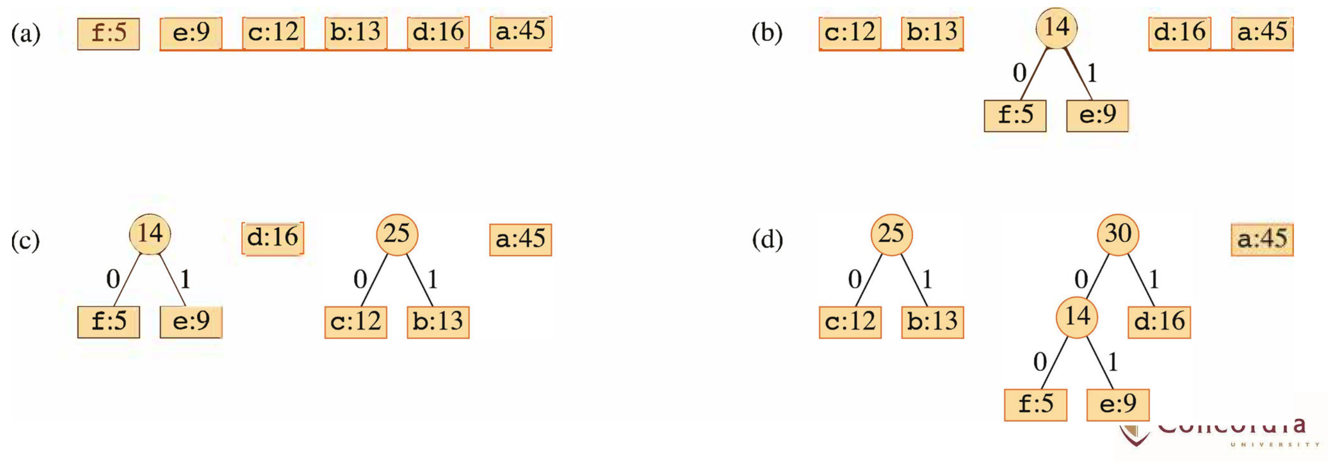
3. Frequency of root of this tree $=$ sum of frequencies of the elements,

4. Put the root of the tree back in the list, in order of frequencies.

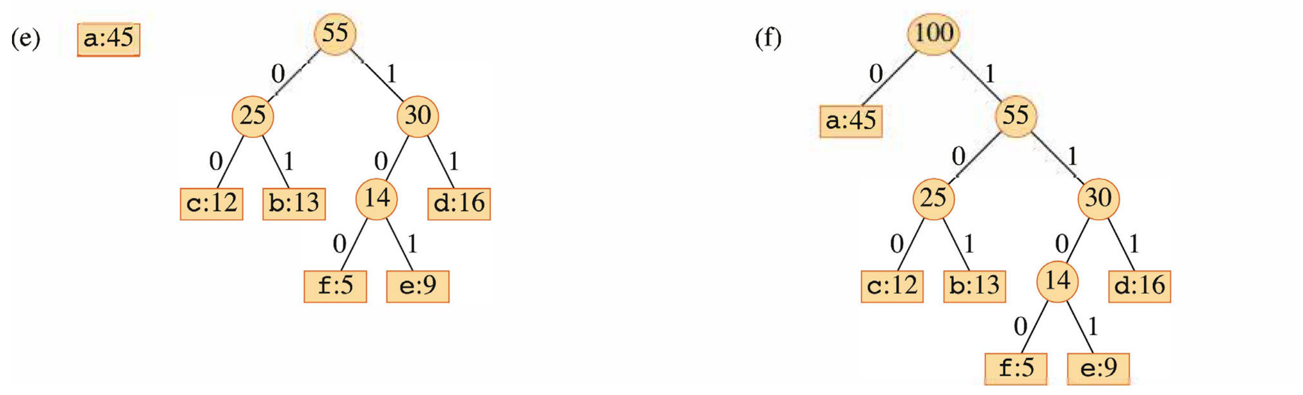
until only one element remains in the list.

The steps of Huffman's algorithm

An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child.



The steps of Huffman's algorithm, cont.



The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.

To improve the run-time, keep nodes in a min priority queue.

```
HUFFMAN($C$)
  \\ $C$ is set of $n$ characters; each $c \in C$ has $c.freq$ as its frequency
  $n \leftarrow |C|$
  $Q \leftarrow C$                                   ▷ Initialize min-priority queue, $Q$
  for $i \leftarrow 1, n-1$ do
      allocate a new node $z$
      $x \leftarrow$ Extract=Min(Q)
      $y \leftarrow$ Extract=Min(Q)
      $z.left \leftarrow x$
      $z.right \leftarrow y$
      $z.freq \leftarrow x.freq + y.freq$
      Insert($Q, z$)        ▷ implementation of Insert determines order of duplicate keys
  return Extract=Min(Q)      ▷ the root of the tree is the only node left
```

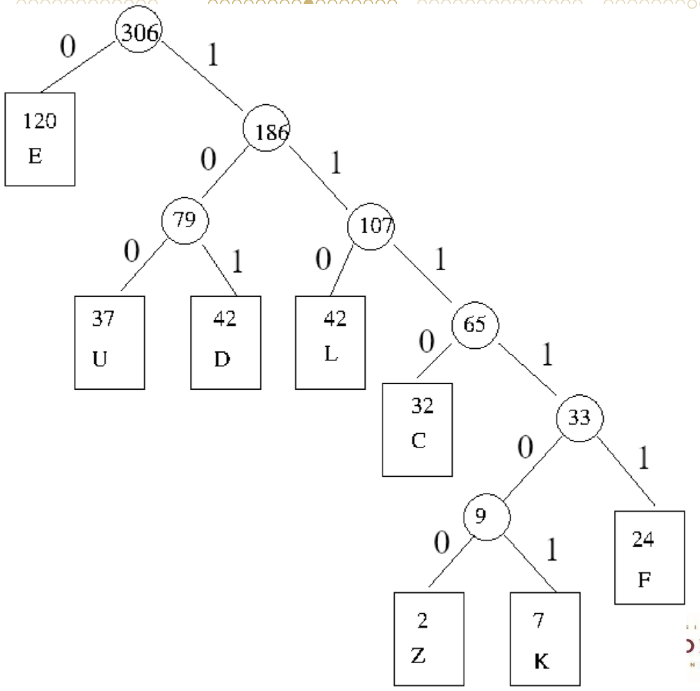
$\boxed{\text{Run-time } = O(n \log n)}$

Example

Z,2 K,7 F,24 C,32 U,37
D,42 L,42 E,120

Codewords:
E=0, U=100, D=101, L=110, ...
F=11111

Code of “FULL” =
11111100110110



Proof of Correctness

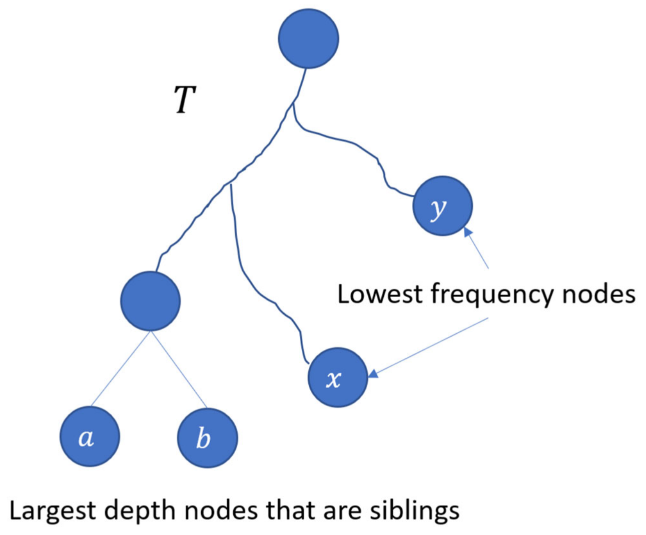
Lemma (Greedy-choice property of Optimal Prefix-free code Trees)

Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Meaning : x and y are leaves of the same internal node.

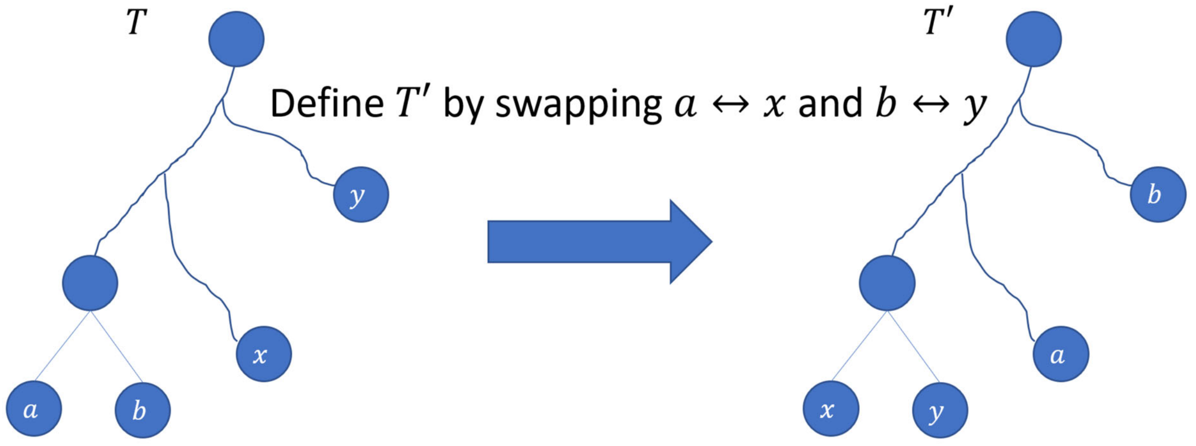
Furthermore, x and y will be at maximum depth in the tree.

Proof Let T be an optimal prefix-free tree. If T does not satisfy the conditions of the Lemma, we show how to modify it to tree T' such that



- T' satisfies the conditions of the Lemma, and
- $B(T') \leq B(T)$

WLOG, we can assume the tree for any optimal prefix-free code is full.
Let a and b be any two characters that are sibling leaves of maximum depth in T. WLOG, assume $a.freq \leq b.freq$ and $x.freq \leq y.freq$.
Therefore,
$x.freq \leq a.freq$ and $y.freq \leq b.freq$, and $d_T(a), d_T(b) \geq d_T(x), d_T(y)$


T
T'
Define T' by swapping $a \leftrightarrow x$ and $b \leftrightarrow y$
y
x
a
b
b
a
x
y

$$\begin{aligned}
B(T') &= B(T) + x.freq{\cdot}d_{T'}(x) + a.freq{\cdot}d_{T'}(a) + y.freq{\cdot}d_{T'}(y) + b.freq{\cdot}d_{T'}(b) \\
&\qquad - x.freq{\cdot}d_T(x) - a.freq{\cdot}d_T(a) - y.freq{\cdot}d_T(y) - b.freq{\cdot}d_T(b) \\
&= B(T) + x.freq{\cdot}d_T(a) + a.freq{\cdot}d_T(x) + y.freq{\cdot}d_T(b) + b.freq{\cdot}d_T(y) \\
&\qquad - x.freq{\cdot}d_T(x) - a.freq{\cdot}d_T(a) - y.freq{\cdot}d_T(y) - b.freq{\cdot}d_T(b) \\
&= B(T) - (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\qquad - (b.freq - y.freq)(d_T(b) - d_T(y)) \\
&\leq B(T)
\end{aligned}$$

Since T is optimal, we have $B(T) \leq B(T')$, which implies $B(T') = B(T)$. Thus, T' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Theorem Huffman's algorithm produces an optimal prefix-free tree.

Proof (by induction on n – number of characters) Base case $n = 1$: obvious.

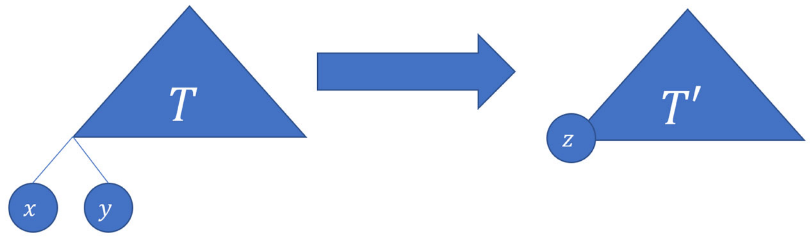
Inductive assumption: assume that Huffman's algorithm produces an optimal prefix-free tree for all inputs with at most n characters for some $n \geq 2$

Inductive step: consider an input C with $n + 1$ characters with x and y being two least frequent characters

Let T be an optimum tree for C with x and y siblings at the lowest level (exists by the Lemma)

Define a new character z with associated frequency $z.freq = x.freq + y.freq$

Let T' be the prefix-free tree for $C \cup \{z\} - \{x, y\}$ obtained by replacing the parent of x, y with z



$$d_T(x) = d_T(y) = d_{T'}(z) + 1 \text{ and } z.freq = x.freq + y.freq$$

Therefore

$$\begin{aligned} B(T') &= B(T) - x.freq{\cdot}d_T(x) - y.freq{\cdot}d_T(y) + z.freq{\cdot}d_{T'}(z) \\ &= B(T) - (x.freq + y.freq) \end{aligned}$$

By induction, Huffman's algorithm finds optimal tree H' for $C \cup \{z\} - \{x, y\}$

$$B(H') \leq B(T')$$

Also, by the algorithm's definition tree H for the original input C satisfies

$$B(H) = B(H') + x.freq + y.freq$$

Combining everything, we obtain:

$$\begin{aligned} B(H) &= B(H') + x.freq + y.freq \\ &\leq B(T') + x.freq + y.freq \\ &= (B(T) - (x.freq + y.freq)) + x.freq + y.freq \\ &= B(T) \end{aligned}$$

Since T is optimal tree for the whole input C and $B(H) \leq B(T)$ the tree output by Huffman's algorithm, H is optimal too.

■

Amortized Analysis (§16.1-16.4)

In usual worst-case analysis of algorithms, for an arbitrary sequence of n operations on a data structure where any operation has a worst case running time T, the total worst case running time is taken to be $n \cdot T$.

However, the actual cost for an operation in the sequence could be lower, since not all operations may require the worst case running time of T.

Amortized analysis can be used to show that the average cost of any operation is small even though certain operations within the sequence might be expensive. Cost of "expensive" operations are "amortized" by cheap ones.

More specifically, in an amortized analysis, the time required for a sequence of *n* operations on a data structure is averaged over all the operations involved. If an "expensive" operation does not happen very often and is in a sequence of "cheap" operations, the average cost can be low.

This is different from average-case analysis since probability is not involved.

Amortized analysis better represents the actual worst case performance of an algorithm.

There are several ways how to conduct amortized analysis.

Two examples will be used as case studies:

1. Stack with **MULTIPOP**, which can pop several objects at once.
2. Binary Counter that counts up from 0 by means of operation **INCREMENT**.

Aggregate Analysis (§16.1)

We show that for all *n*, a sequence of *n* operations takes *worst case* time $T(n)$ in total. Thus, cost per operation is $T(n)/n$.

Example: Stack Operations with additional **MULTIPOP** operation.

```
MULTIPOP($S, k$)
  while not Stack-Empty($S$) & $k > 0$ do
      POP($S$)
      $k \leftarrow k - 1$
```

The cost of a single **MULTIPOP** $= \min\{k, length(S)\}$

$length(S)$ is the number of elements in the stack.

Aggregate Analysis:

Now consider a sequence of n **PUSH** and **MULTIPOP**s on initially empty stack.

Since $length(S) \leq n$ and the cost for a **MULTIPOP** is $O(n)$, the typical worst case cost could be $O(n \cdot n)$.

However, this analysis is overestimating the cost of **MULTIPOP**s.

In total, we cannot pop more than what was pushed in the stack.

We pushed in at most n items using **PUSH**,

so total cost of all **MULTIPOP**s is $O(n)$.

Thus the cost is $O(n) + O(n) = O(n)$ for all operations, so we can say any one operation on average is $O(n)/n = O(1)$.

The above is the worst case!

Example: Binary counter

```
INCREMENT(A, k)
    $i \leftarrow 0$
    while $i < k$ & $A[i] = 1$ do
        $A[i] \leftarrow 0$
        $i \leftarrow i + 1$
    if $i < k$ then
        $A[i] \leftarrow 1$
```

What is the cost of repeating **INCREMENT** n times?

Brief inspection:
Cost of each increment is linear in the number of bits we flip,
The number of bits goes up so it is $\sum_{i=1}^{\log n} i = O(n \log n)$.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Aggregate Analysis:

Not every bit flips each time.

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	$\lfloor n/2 \rfloor$
2	1/4 the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

So the total number of flips in n operations is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor = n \sum_{i=0}^{k-1} \left\lfloor \frac{1}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \left\lfloor \frac{1}{2^i} \right\rfloor = n \cdot \frac{1}{1 - 1/2} = 2n$$

So on average the cost is 2 per operation, i.e., $O(1)$.

Important:
we consider that there is no mix of increments and decrements.

Accounting Method (§16.2)

We assign to operations cost that differ from the actual cost.

Some are charged more than the actual cost,
some are charged less than the actual cost.

The amount charged is called **amortized cost**.

What we need is that everything balances out at the end, i.e. the total of amortized charges must be at least as high as the actual cost. The extra is called credit.

Sometimes this may allow easier analysis:
by over-charging for some operations that are easy to count we don't need to charge anything to operations that are more complicated to count.

Example: Stack Operations with **MULTIPOP**.

Actual cost	Let us assign the following *amortized costs*
PUSH 1	**PUSH** \$2
POP 1	**POP** 0
MULTIPOP ... $\min\{k, s\}$	**MULTIPOP** ... 0

Now consider any sequence of n **PUSH**, **POP** and **MULTIPOP**s on initially empty stack.

Each **PUSH** is over-charging the cost by \$1. Thus, we "pre-pay" for the eventual **POP** or **MULTIPOP** when push is done.

Thus any sequence of **PUSH**, **POP** and **MULTIPOP** operations has amortized cost $2\cdot$ # **PUSH**. If at most n **PUSH** operations are done, the total amortized cost is $\leq 2n$ and the total actual cost is $O(n)$.

Example: Binary counter.

Let the amortized cost of setting a bit from 0 to 1 be \$2.

Thus we charge \$1 to set the bit to 1 and we have completely pre-payed the cost of \$1 of the eventual change of the same bit back 1 to 0.

```
INCREMENT($A, k$)
  $i \leftarrow 0$
  while $i < k$ & $A[i] = 1$ do
      $A[i] \leftarrow 0$
      $i \leftarrow i + 1$
  if $i < k$ then
      $A[i] \leftarrow 1$
```

So in the loop all operations of setting a bit to 0 are pre-paid and cost nothing.

Thus the amortized cost of a single **INCREMENT** is $2 = O(1)$ since only one 0 bit is flipped to a 1 bit.

A sequence of n **INCREMENT**s is $O(n)$.

Potential Method (§16.3)

In the potential method of amortized analysis, we represent the prepaid work as "*potential energy*" that can be released later for the cost of future operations.

The potential is associated with the data structure as a whole rather than with specific objects.

We have a data structure D.

D_i ... the data structure D after applying ith operation to it.

Initially, we have D_0.

A potential function Φ maps the data structure D_i to a real number $\Phi(D_i)$, the associated potential.

The **amortized cost** $\hat{c}_i$ of the ith operation with respect to the actual cost c_i and Φ is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

i.e. the actual cost + difference in potential due to the operation.

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

If $\Phi(D_n) \geq \Phi(D_0)$ then the total amortized cost is at least as great as the total actual cost, $\sum_{i=1}^{n} c_i$.

Example: Stack Operations with **MULTIPOP**.

Define the potential to be the *number of objects in the stack*

$$\Phi(D_n) \geq \Phi(D_0) = 0$$

For the cost of **PUSH** on a stack with s objects, the difference in potential is

$$\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$$

So the amortized cost of **PUSH** is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2$$

Similarly, the amortized cost of **POP** is 0.

Cost of **MULTIPOP**(S, k) on a stack with s objects:

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1})) = \min\{s, k\} - \min\{s, k\} = 0$$

Example: Binary counter.

Define the potential after *i*th **INCREMENT** to be the *number of* 1*'s in the counter* = b_i

Suppose that the *i*th **INCREMENT** resets t_i bits from 1 to 0.

The actual cost is at most $t_i + 1$ since in addition to resetting t_i bits from 1 to 0, it sets at most one bit from 0 to 1.

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

Amortized cost:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all *i*, the total amortized cost of a sequence of *n* **INCREMENT**s is an upper bound on the actual cost and so an upper bound on *n* **INCREMENT**s is $O(n)$.

Dynamic Tables (§16.4)

If we use a table T implemented as an array and the table becomes too small, we can

1. allocate a new larger table T',
2. copy the old table T into T', and
3. delete T.

Call it *table expansion*. What is the cost of this table expansion?

Common strategy:

When full, double the existing table size

We define the *load factor* of a table $\alpha = \frac{\text{the number of items}}{\text{table size}}$

```
TABLE-INSERT($T, x$)
    if $T.size = 0$ then
        allocate $T.table$ with 1 slot
        $T.size \leftarrow 1$
    if $T.num = T.size$ then
        allocate new-table with $2 \cdot T.size$ slots
        insert all items in $T.table$ into new-table
        free $T.table$
        $T.table \leftarrow$ new-table
        $T.size \leftarrow 2 \cdot T.size$
    insert $x$ into $T.table$
    $T.num \leftarrow T.num + 1$
```

We analyze the run-time of this in terms of the number of basic insertions.

It is assumed that to create, delete an array is constant time regardless of the size of array. (commonly true)

Since the copying is costly, the cost of table insert could be n, and we might say that the cost of n insertions is $O(n^2)$.

Aggregate Analysis

Cost c_i of i'th insertion is:

i if i is a power of 2,
1 otherwise.

The total cost of n operations is

$$\sum_{i=1} nc_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j = n + \frac{2^{\lfloor \log n \rfloor + 1} - 1}{2 - 1} < n + 2n = 3n$$

So aggregate analysis says if we charge each insert operation an amortized cost of 3 units, we "pay" for everything.

Accounting Method

Charge amortized cost of \$3 per insertion of x.

- \$1 pays for x's insertion.
- \$1 pays for x to be moved in the future.
- \$1 pays for some other item to be moved.

Let $num =$ items stored, $size =$ allocated size.

Suppose the table has just expanded, $size = num = m$ just before the expansion, and $size = 2m$ and $num = m$ just after the expansion (before adding the next item). The next expansion occurs when $size = num = 2m$.

Assume that the expansion used up all the credit, so that there's no credit stored after the expansion. Will expand again after another m insertions.

Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.

Have \$2m of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!

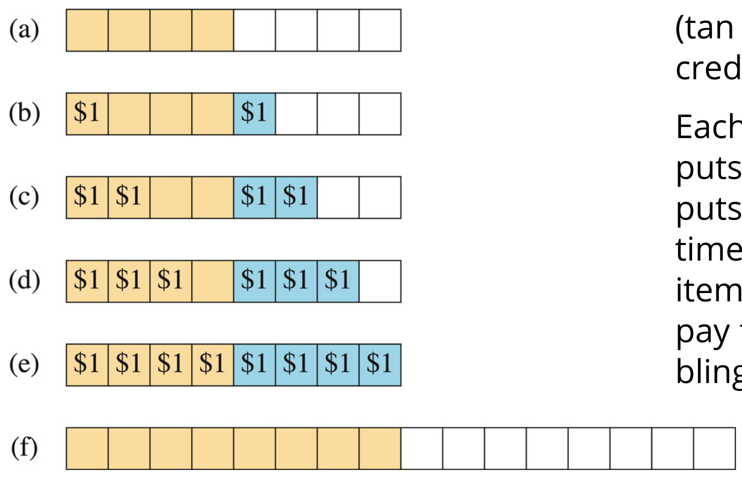


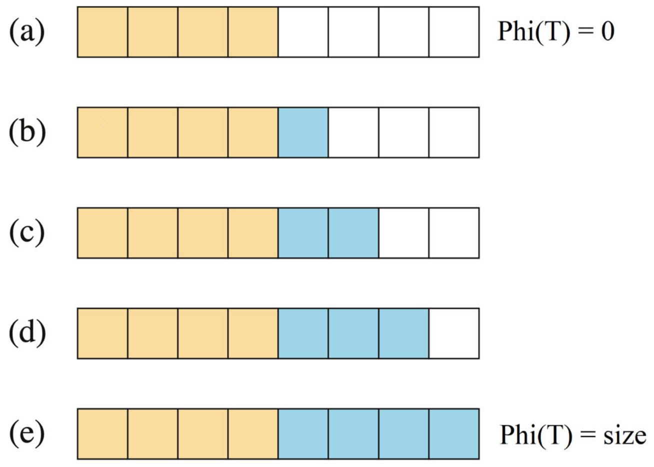
Table with *size* $=$ 8 and *num* $=$ 4 (tan squares), so that it has no stored credit.

Each insertion (blue squares) costs \$1, puts \$1 on the item just inserted, and puts \$1 on some other item. By the time table fills (*num* $=$ *size* $=$ 8), each item has \$1 stored on it $\Rightarrow$ enough to pay to reinsert all the items after doubling the table size.

Thus the amortized analysis shows that the insertion is $O(1)$ in this case.

Potential Method

Think of potential being 0 just after expansion — when *num* $=$ *size*$/2$.
(Just as the accounting method has no stored credit just after an expansion.)
As elementary insertions occur, the table needs to build enough potential to pay to reinsert all the items at the next expansion — when *num* $=$ *size*, which is after *size*$/2$ insertions. The potential needs to be *size* at that time.



$\Rightarrow$ Over *size*$/2$ insertions, potential needs to go from 0 to *size*

$\Rightarrow$ Potential increase per insertion is

$$\frac{size}{size/2} = 2$$

Use the potential function

$$\Phi(T) = 2(T.num - T.size/2)$$

The potential equals 0 just after expansion, when $T.num = T.size/2$

The potential equals $T.size$ when the table fills, when $T.num = T.size$

Initial potential is 0, and the potential is always non-negative $\rightarrow$ sum of the amortized costs gives an upper bound on the sum of the actual costs.

$$\begin{aligned} \Phi_i &= \text{potential after the } i\text{th operation },\\ \Phi_i - \Phi_{i-1} &= \text{change in potential due the } i\text{th operation },\\ \hat{c}_i &= c_i + (\Phi_i - \Phi_{i-1}) \end{aligned}$$

When the *i*th insertion does not trigger expansion

$$c_i = 1 \text{ and } (\Phi_i - \Phi_{i-1}) = 2 \Rightarrow \hat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 2 = 3$$

When the *i*th insertion does trigger expansion (table is full)

$$\begin{aligned} num_i &= \text{number of items in the table after the } i\text{th operation },\\ size_i &= \text{size of the table after the } i\text{th operation }. \end{aligned}$$

Before insertion:
$size_{i-1} = num_{i-1} = i - 1 \Rightarrow$

$$\begin{aligned} \Phi_{i-1} &= 2(size_{i-1} - size_{i-1}/2)\\ &= size_{i-1}\\ &= i - 1 \end{aligned}$$

After expansion: $\Phi = 0$.
After inserting new item: $\Phi_i = 2$.
$\Rightarrow (\Phi_i - \Phi_{i-1}) = 2 - (i-1) = 3 - i$.
Actual cost $c_i = i$ ($i - 1$ reinsertions and 1 insertion of new item)
$\Rightarrow$ amortized cost is $\hat{c}_i = c_i + (\Phi_i - \Phi_{i-1})$
$\Rightarrow \hat{c}_i = i + (3 - i) = 3$

So the amortized cost of an insertion is $O(1)$.

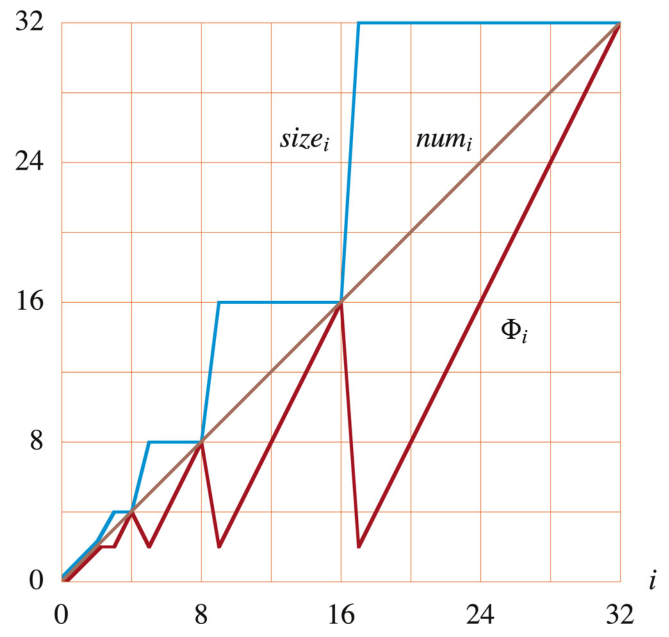


Table expansion and contraction

Contraction:
If the load factor α of the table is too small, shrink it.

We cannot use the same strategy for table contraction and table expansion:

If we contract when α is 1/2, we could keep on expanding and contracting repeatedly, which is very costly.

Good strategy:
Shrink the table size to half size when α is 1/4.

This way we avoid frequent expansion and contraction within a sequence of a few instructions (e.g., insert, delete, insert, insert, delete, delete, insert, insert, . . .)

Again, we can show that operations remain $O(1)$ using amortized analysis.

Potential Method for table expansion and contraction

Overall potential function:

$$\Phi(T) = \begin{cases} 2(\text{T.num - T.size/2}) & \text{if } \alpha(T) \geq 1/2 \,, \\ \text{T.size/2 - T.num} & \text{if } \alpha(T) < 1/2 \,. \end{cases}$$

Initially, $num_0 = 0, size_0 = 0, \Phi_0 = 0$

See textbook (§16.4.2) for detailed proof that the amortized cost for both insertion and deletion is $O(1)$

Example

Consider using a singly linked list to maintain a set S of n distinct integers that supports the following two operations:

1. INSERT(x, S): insert integer x into S.
2. REMOVE-BOTTOM-HALF(S): remove the smallest $\lceil n/2 \rceil$ integers from S.

First, consider the (worst-case) cost c_i of the two operations.

To implement INSERT(x, S), we append the new integer to the end of the linked list. This takes $c_i = \Theta(1)$ time.

To implement REMOVE-BOTTOM-HALF(S), we use the median finding algorithm taught in class to find the median number, and then go through the list again to delete all the numbers smaller or equal than the median. This takes $c_i = \Theta(n)$ time.

Example, cont.

Let's consider using the potential method to perform an amortized analysis of INSERT(x, S) and REMOVE-BOTTOM-HALF(S).

Suppose the runtime of REMOVE-BOTTOM-HALF(S) is bounded by dn for some constant d. For amortized analysis, use $\Phi = 2dn$ as our potential function where the list has n integers.

Therefore, the amortized cost of an insertion is

$$\hat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = 1 + 2d(n+1) - 2dn = 1 + 2d = \Theta(1)$$

The amortized cost of REMOVE-BOTTOM-HALF(S) is

$$\hat{c}_i = c_i + (\Phi_i - \Phi_{i-1}) = dn + (2d(n/2) - 2dn) = dn + (-2d \times \frac{n}{2}) = 0$$

The amortized cost for both operations is $\Theta(1)$