CMPT231

Lecture 9: ch16

Greedy Algorithms

2 Peter 3:10-12 (NASB)

But the **day of the Lord** will come **like a thief**, in which […] the **earth** and its **works** will be burned up.

Since all these things are to be **destroyed** in this way, **what sort** of people ought you to be

in **holy conduct** and **godliness**, **looking** for and **hastening** the coming of the day of God

Outline for today

- **Greedy** algorithms
- **Activity selection** problem
  - Optimal substructure
  - Proof of optimality of greedy
  - Greedy solution
- **List merging** problem
  - Proof of optimality of greedy
- **Huffman** coding
- **Knapsack** problem: fractional and 0-1
- Optimal offline **caching**

Greedy algorithms

- A special case of **dynamic programming**
  - At each decision point, choose **immediate** gains
- Relies on **greedy choice** property:
  - Locally optimal choices $\Rightarrow$ global optimum
- **Not** all problems have greedy choice property!
- **Hybrid** strategies use large jumps to get to right "hill"
- Then use greedy **hill-climbing** to get to the top

![Saddle point between maxima, [Wikin
(https://commons.wikimedia.org/wiki/File%3ASaddle_Poin
(static/img/Saddle_Point_between_maxi

Problem-solving outline

- Describe optimal **substructure** (e.g., via recurrence)
- Convert to naive **recursive** solution
  - Could then convert to dynamic programming
- Use **greedy choice** to simplify recurrence
  - So only one subproblem remains
  - Don’t have to iterate through all subproblems
  - Need to **prove** greedy choice yields global optimum
- Convert to **recursive** greedy solution
- Convert to **iterative** (bottom-up) greedy solution

Outline for today

- Greedy algorithms
- **Activity selection problem**
  - **Optimal substructure**
  - Proof of optimality of greedy
  - Greedy solution
- List merging problem
  - Proof of optimality of greedy
- Huffman coding
- Knapsack problem: fractional and 0-1
- Optimal offline caching

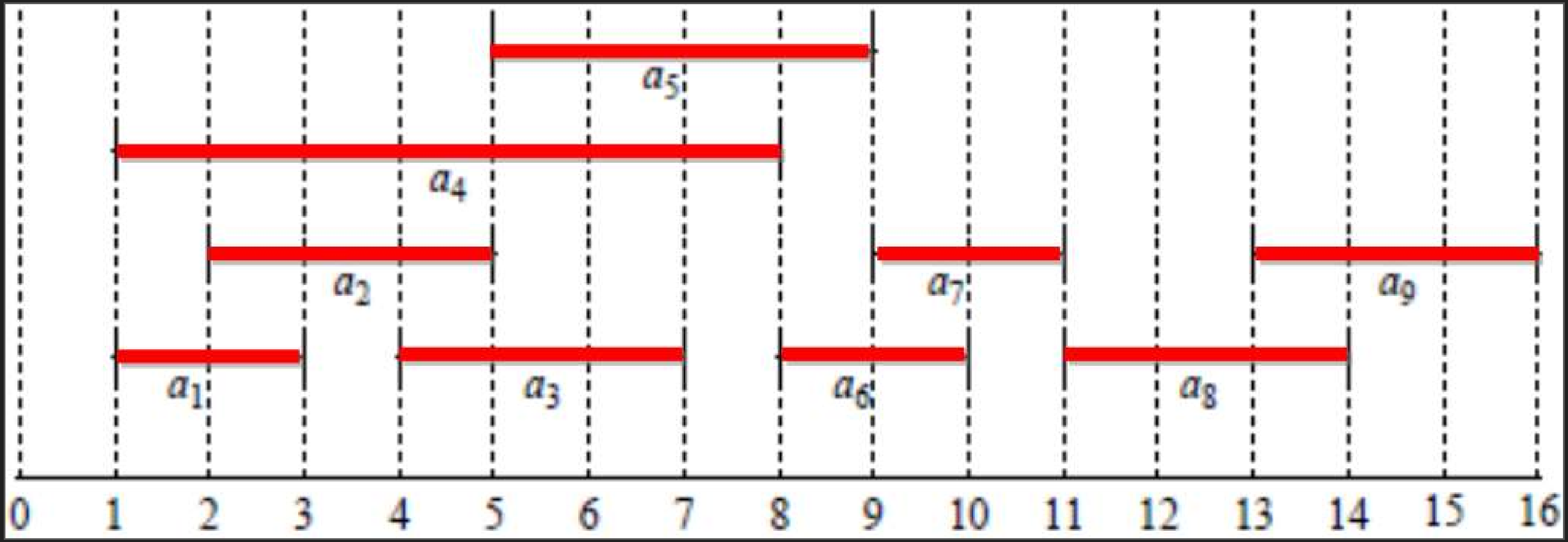
Example: activity selection

- **Activities** $S = \{a_i\}_1^n$:
  each require exclusive use of some shared resource
  - e.g., database table, communication bus, conference room
- Each activity has **start**/finish times $[s_i, f_i)$
  - Input is **sorted** by finish time
- Task: **maximise** num activities that can be completed
  - i.e., find **largest** subset of S with **non-overlapping** activities

i	1	2	3	4	5	6	7	8	9
s	1	2	4	1	5	8	9	11	13

ActSel: example

i	1	2	3	4	5	6	7	8	9
s	1	2	4	1	5	8	9	11	13
f	3	5	7	8	9	10	11	14	16



ActSel: task substructure

- Let $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
  - all activities that **start** after f_i and **finish** before s_j
- Any activity in S_{ij} will be **compatible** with:
  - Any activity that **finishes** no later than f_i
  - Any activity that **starts** no earlier than s_j
- Let A_{ij} be a **solution** for S_{ij}:
  - i.e., largest mutually-compatible subset of S_{ij}
- Choose an activity $a_k \in A_{ij}$ and **partition** A_{ij} into
  - $A_{ik} = A_{ij} \cap S_{ik}$: those that finish **before** a_k starts
  - $A_{kj} = A_{ij} \cap S_{kj}$: those that start **after** a_k finishes

ActSel: prove opt substr

- **Claim**: A_{ik} and A_{jk} are **optimal** solutions for S_{ik}, S_{kj}
- **Proof** (for A_{ik}): assume **not**:
  - Let B_{ik} be a **better** solution:
  - So B_{ik} has non-overlapping elts, and $|B_{ik}| > |A_{ik}|$
  - Then $B_{ik} \cup \{a_k\} \cup A_{kj}$ would be a **solution** for S_{ij}
  - Its size is **larger** than $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - This **contradicts** the assumption that A_{ij} was **optimal**
- Hence our optimal **substructure** is:
  - **Split** on a_k,
  - **Recurse** twice on the two subproblems S_{ik} and

ActSel: naive recursive

- Let c(i, j) be the **size** of an optimal solution for S_{ij}:
  - **Splitting** on a_k yields c(i, j) = c(i, k) + 1 + c(k, j)
  - Which **choice** of a_k is the best? Try **all** of them
- **Recurrence**:

$$c(i, j) = \begin{cases} 0 & \text{if} \quad S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \left(c(i, k) + 1 + c(k, j) \right) & \text{if} \quad S_{ij} \neq \emptyset \end{cases}$$

- **Dynamic programming** implementation:
  - Fill in 2D **table** for c(i, j), bottom-up
  - Auxiliary table to store **solutions** A_{ij}
- But for this problem, we can do even **better**!

Outline for today

- Greedy algorithms
- Activity selection problem
  - Optimal substructure
  - **Proof of optimality of greedy**
  - **Greedy solution**
- List merging problem
  - Proof of optimality of greedy
- Huffman coding
- Knapsack problem: fractional and 0-1
- Optimal offline caching

ActSel: greedy choice

- Which **choice** of a_k leaves as **much** as possible of the shared resource available for the other activities?
  - The one which **finishes** the earliest
  - Since input is **sorted** by finish time, just take the **first** activity
- Let $S_i = \{a_k \in S : f_i \leq s_k\}$: those that **start** after a_i finishes
- **Simplified** recurrence: to find optimal subset of S_i,
  - Choose the **first** activity in S_i: call it a_j
  - Recurse on the **remainder**: S_j
  - Don't need to **iterate** over all choices of $a_k \in S_i$
- We need to **prove** this greedy choice is **optimal**

ActSel: prove greedy

- **Cut-and-paste** style proof:
- Let A_i be any **optimal solution** for S_i
- **Modify** A_i to fit greedy strategy
  - Let a_j have **earliest** finish time in S_i
    - This would be the greedy choice
  - Let a_k have **earliest** finish time in A_i
    - This is the choice in the given **optimal** solution
  - **Swap** them: let $B_i = A_i - \{a_k\} \cup \{a_j\}$
- Show the modified solution B_i is just as **optimal**:
  - **Size** $|B_i|$ is same as $|A_i|$
  - Activities are **non-overlapping**: $f_j \le f_k$

ActSel: recursive greedy

- **Input**: arrays `s[], f[]`, sorted by `f[]`
  - Add a **dummy** entry `f[0]=0` so that $S_0 = S$
- For each **subproblem** S_i:
  - **Skip** over any activities that overlap with a_i
  - Choose the **first** activity a_j that doesn't overlap
  - **Recurse** on the remainder
- **Complexity**?

```
def ActivitySelection( s, f, i ):
  for j in i+1 .. length( f ):
    if ( s[ j ] >= f[ i ] ):


# compatible w/ a_j?
      return [ j ] + ActivitySelection( s, f, j )   # concatenate
  return NULL
```

ActSel: iterative greedy

- Easy to convert **tail-recursion** to **iteration**
- **Complexity**: $\Theta(n)$
  - or $\Theta(n \lg n)$ if need to **pre-sort** `f[]`

```
def ActivitySelection( s, f ):
  A = [ 1 ]
  i = 1
  for j in 2 .. length( f ):
    if ( s[ j ] >= f[ i ] ):    # compatible w/ a_j ?
      A = A + [ j ]
     # append to list
      i = j
  return A
```

i	1	2	3	4	5	6	7	8	9
s	1	2	4	1	5	8	9	11	13
f	[illegible]	5	7	[illegible]	[illegible]	10	11	14	[illegible]

Greedy vs dynamic programming

- Greedy-solvable problems are a **subset** of dynamic programming-solvable problems
  - **Not** all problems have greedy property
- Dynamic programming fills in table **bottom-up**
  - Greedy choice is done **top-down**
- Dyn prog **choice** requires solutions to **all** subproblems
  - Greedy choice can be made **before** doing the (single) subproblem
- To **prove** greedy property:
  - Assume an optimal solution
  - Modify it to include the greedy choice

Optimising for greedy choice

- Often, greedy choice is easier if input is **pre-processed**
- E.g., **sorting** activities by finish time
  - Then the greedy **choice** can be made in O(1) each time
  - The **pre-sorting** takes O(n lg n)
- If input is **dynamically** generated:
  - Can't **sort** whole list in advance, but we can
  - Use **priority queue** to pop the current most optimal choice

Outline for today

- Greedy algorithms
- Activity selection problem
    - Optimal substructure
    - Proof of optimality of greedy
    - Greedy solution
- **List merging problem**
    - **Proof of optimality of greedy**
- Huffman coding
- Knapsack problem: fractional and 0-1
- Optimal offline caching

List merging

- Given: **lists** of various lengths, $l_1 < l_2 < \ldots < l_n$
  - Want: **sequence** of lists to merge, minimising total merge cost
- **Merging** two lists \`l\_i, l\_j\` costs \`l\_i+l\_j\`
  - and creates a new list of length \`l\_i+l\_j\`

![List merge / Huffman](static/img/HuffmanCodeAlg.png)

- **Applications**: merge sort

List merge: greedy strategy

- Always select the two **shortest** lists to merge
  - Merging creates a **new** list, that is also **pushed** with other lists
  - **Iterate** on the new set of lists
- $\Rightarrow$ what **data structure** to use?
  - Hold a **set** of keys
  - Quickly return the **smallest** key
- $\Rightarrow$ **Complexity** of finding optimal schedule for n lists?

List merge: notation

- Represent a merge **schedule** using a binary **tree**:
  - Input lists are at **leaves**
    - Keys are **lengths** of lists
  - Keys for **internal** nodes are **sum** of children’s keys
- Total merge **cost** is sum of all interior nodes:
  - $= \sum_{i=1}^{n} d_i l_i$, where d_i is **depth** of leaf i in tree
- Note: any leaf of **maximal** depth must have a **sibling**
  - Sibling must also be a **leaf**
  - Otherwise would not be **maximal** depth

List merge: proof outline

- **Prove** that greedy strategy gives an optimal solution:
  - Use **induction** on number n of lists
- For inductive step, use a **cut-and-paste** style proof:
  - Pick an arbitrary **optimal** (not necessarily greedy) solution
  - **Modify** it to fit the greedy strategy, and
  - Show the modified solution is **no worse** than original solution
  - Apply **inductive hypothesis** on set of n-1 lists
  - Thus greedy on n lists is **no worse** than the modified solution
  - Which is **no worse** than the original, optimal

List merge: proof

- Let T be the tree for **any** optimal solution
  - Let u and v be sibling **lists** of maximal depth, $d_{\max}$ (wlog, u ≤ v)
- Consider two **smallest** lists, l_1, l_2, of depth d_1, d_2
  - **Greedy** strategy would put these two at maximal depth
- **Swap** u $\leftrightarrow$ l_1, and v $\leftrightarrow$ l_1: call the modified tree T’
  - First merge in T’ is same as **greedy**
  - Remaining merges in T’ no better than greedy, by **inductive hyp**
- How does this affect the **total merge cost**?

cost(T’) - cost(T) =

Outline for today

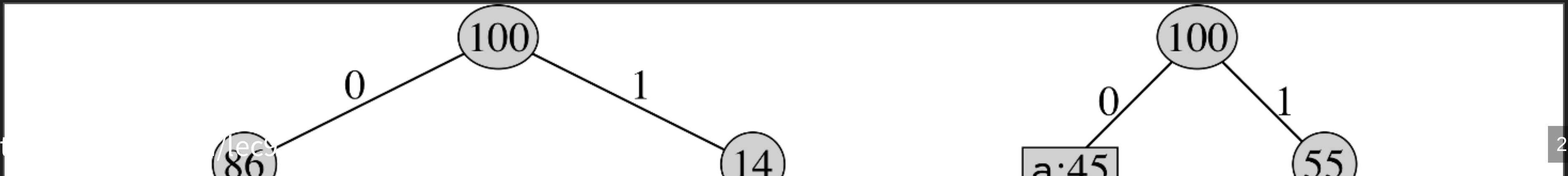
- Greedy algorithms
- Activity selection problem
    - Optimal substructure
    - Proof of optimality of greedy
    - Greedy solution
- List merging problem
    - Proof of optimality of greedy
- **Huffman coding**
- Knapsack problem: fractional and 0-1
- Optimal offline caching

Encoding

- Given a **text** with known character set
  - **Encode** each character with a unique codeword in binary
- **Fixed-length** code: all codewords same length
  - “`cafe`“ ⇒ 010 000 101 100
- **Variable-length** code: some codes use fewer bits
  - “`cafe`“ ⇒ 100 0 1100 1101
  - **Compression**: more frequent chars get shorter codes
- **Prefix** code: no code is a prefix of another
  - Makes **parsing** unique: don’t need delimiters
  - “`cafe`“ ⇒ 100011001101

Binary code trees

- **Prefixes** are nodes, **characters** are at leaves
  - Requires encoding to be a prefix code
- **Decoding** strings = a series of walks down the tree
  - **Cost** of a character c = its depth d_c in the tree
  - **Fixed-length** code $\Rightarrow$ all leaves at **same level**
- **Total cost** of encoding a text using a given tree: $\sum (f_c d_c)$
  - where f_c is the **frequency** of character c in the text



Huffman coding

- Build tree **bottom-up**:
  - Start with the two **least**-common characters
  - **Merge** them to make a new subtree with **combined** freq
- Sounds **familiar**?
- Use **min-priority queue** to manage the greedy choice

```
def huffman( chars ):
  Q = new MinQueue( chars )
  for i in 1 .. length( chars ) - 1:
    z = new Node()
    z.left = Q.popmin()
    z.right = Q.popmin()
    z.freq = z.left.freq + z.right.freq
    Q.push( z )
  return Q.popmin()
```

Outline for today

- Greedy algorithms
- Activity selection problem
  - Optimal substructure
  - Proof of optimality of greedy
  - Greedy solution
- List merging problem
  - Proof of optimality of greedy
- Huffman coding
- **Knapsack problem: fractional and 0-1**
- Optimal offline caching

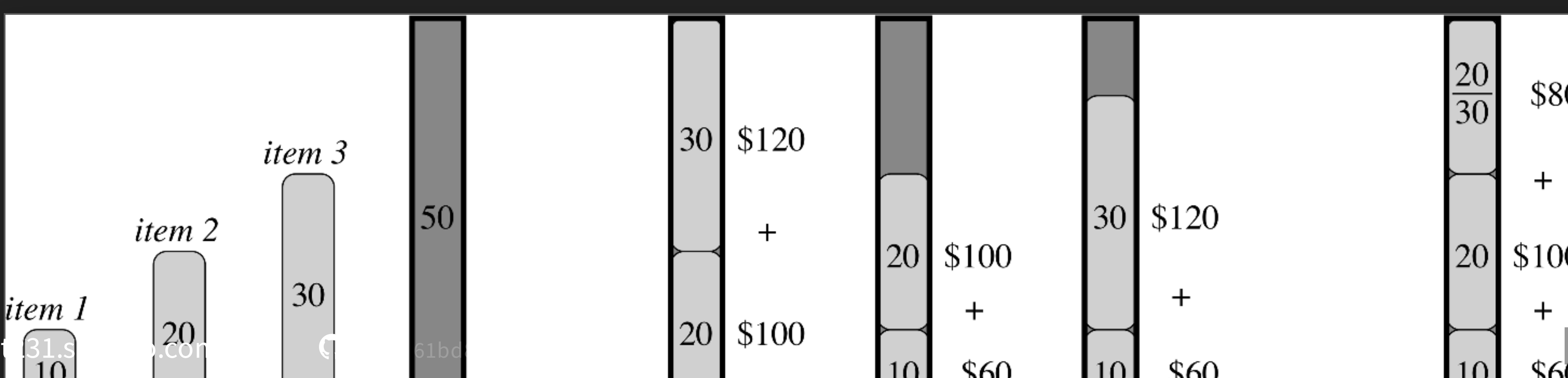
Knapsack problem

- **Fractional** knapsack problem:
  - n items, each with **weight** w_i and **value** v_i
  - Maximise total **value**, subject to total **weight** limit W
  - Can take **fractions** of an item (think liquids)
- **Applications**: stock portfolio selection, spacecraft packing, cargo ships, sheet metal cutting
- **Greedy** solution: sort items by value-to-weight ratio
  - Greedy **choice**: take item of largest $\frac{v_i}{w_i}$
- **Final** spot may be filled with a fractional item

```
def FractionalKnapsack( v, w, W ):
  while totWeight < W:
```

0-1 knapsack

- Knapsack problem, but **no** fractional items allowed
- Greedy strategy **no longer** works!
    - Locally-optimal choices made early on **lock** us out of later globally-optimal choices
- Still can solve with **dynamic programming** (#16.2-2)
- **Knapsack cryptography** relies on complexity of solving



Outline for today

- Greedy algorithms
- Activity selection problem
    - Optimal substructure
    - Proof of optimality of greedy
    - Greedy solution
- List merging problem
    - Proof of optimality of greedy
- Huffman coding
- Knapsack problem: fractional and 0-1
- **Optimal offline caching**

Caching

- Cache has **capacity** to store k items
  - Input is a sequence of m **item requests** $\{d_i\}_1^m$
- Cache **hit** if item already in cache when requested
  - If cache **miss**, need to **evict** an item from cache
  - Then bring **requested** item into cache
- **Task**: eviction schedule to **minimise** evictions
- E.g., k=2, intial cache = ab, requests: a b c b c a b
  - **Optimal** schedule has only 2 evictions:

request	**a**	**b**	**c**	**b**	**c**	**a**	**b**
cache1	a	a	**c**	c	c	**a**	a
cache2	b	b	b	b	b	b	b

Greedy offline caching

- Assume entire request sequence is known in advance
- **LIFO** / **FIFO**: evict most (least) recently added item
- **LRU**: evict item whose most recent access is the earliest
- **LFU**: evict item least frequently accessed

request	**a**	**d**	**a**	**b**	**c**	**e**	**g**
cache1	**a**	a	a	a	a	a	?
cache2	w	w	w	**b**	b	b	?
cache3	x	x	x	x	**c**	c	?
cache4	y	**d**	d	d	d	d	?

Farthest-in-future

- Since this is **offline**, we can **peek** into the future
- **Farthest-in-future** algorithm ("clairvoyant"):
  - Evict item whose next request is the **farthest** in the future
- **Provably** optimal offline caching strategy (Bélády 1966)
  - Proof is also a cut-and-paste greedy proof
- Useful perspective to guide us for **online** algorithms
  - **LRU** is farthest-in-future with time run backwards!
  - **LIFO** can be arbitrarily bad
- Caching is one of comp sci's **hardest** real-world problems

Outline for today

- **Greedy** algorithms
- **Activity selection** problem
  - Optimal substructure
  - Proof of optimality of greedy
  - Greedy solution
- **List merging** problem
  - Proof of optimality of greedy
- **Huffman** coding
- **Knapsack** problem: fractional and 0-1
- Optimal offline **caching**