

CSCI 4470/6470 Algorithms, Spring 2017

Liming Cai

Department of Computer Science, UGA

Syllabus: <http://cobweb.cs.uga.edu/~cai/courses/algo/2017Spring/>

January 17, 2017

An Introduction to the Introduction

An Introduction to the Introduction



Round 1		Round 2		Round 3		Round 4		Quarterfinals		Semifinals	
1.	DJOKOVIC, Novak SRB	[1]	N. DJOKOVIC [1]	6-3 1 5-7 6-2 1 6-1	N. DJOKOVIC [1]						
2.	JANOWICZ, Jerry POL.		J. VESELY	Walkover							
3.	VESELY, Jiri CZE										
4.	MYNENI, Saketh IND	(Q)	G. PELLA	7-6(5) 1-6 6-1 2-6 2-7 6-5							
5.	FRATANGOLO, Birom USA	(W)	M. YOUZHNY	6-2 1 6-1 7-6(3)							
6.	PELLA, Guido ARG										
7.	YOUZHNY, Mikhail RUS										
8.	KLIZAN, Martin SVK	[28]	J. ISNER [20]	6-2 1 6-1 6-1							
9.	ISNER, John USA	[20]									
10.	TIAFOE, Frances USA	(W)	S. DARGIS	3-6 1-6 1 7-6(5) 1-6 2-7 6-2							
11.	THOMPSON, Jordan AUS										
12.	DARGIS, Steve BEL	(Q)	K. EDMUND	6-3 1 6-4 1 6-7(10) 1-6 3							
13.	LACKO, Lukas SVK	(W)	F. ESCOBEDO	5-7 3-6 1 7-6(5) 1-6 4-7 5							
14.	ESCOBEDO, Ernesto USA	(W)	K. EDMUND	6-4 1-6 1 6-4 1 6-3 1 6-4							
15.	EDMUND, Kyle GBR										
16.	GASQUET, Richard FRA	[13]	J. TSONGA [9]	6-2 1 6-2 1 6-3							
17.	TSONGA, Jo-Wilfried FRA	[9]									
18.	ANDREOZZI, Guido ARG	(Q)	J. TSONGA [9]	6-3 1 6-4 1 6-4							
19.	DUCKWORTH, James AUS	(W)	J. DUCKWORTH	6-4 1 3-6 1 6-3 1 6-4							
20.	HAASE, Robin NED										
21.	POSPISIL, Vasek CAN	(L)	K. ANDERSON [23]	7-6(3) 1 6-4 1 6-4							
22.	KOVALIK, Jozef SVK										
23.	NISHIOKA, Yoshihito JPN	[23]	J. SOCK [26]	7-6(3) 1 5-7 3-6 1 1-6 1 6-4							
24.	ANDERSON, Kevin RSA	[26]									
25.	SOCK, Jack USA										
26.	FRITZ, Taylor USA	(Q)	M. ZVEREV	6-1 1 6-6 1 6-2							
27.	ZVEREV, Mischa GER										
28.	HERBERT, Pierre-Hugues FRA		S. STAKHOVSKY	6-4 1 7-6(6) 4-6 1 6-0							
29.	STAKHOVSKY, Sergiy UKR										
30.	ELIAS, Gastao POR		M. CILIC [7]	6-1 3-6 1 6-2 1 6-3 1 7-6(4)							
31.	DUTRA SILVA, Rogério BRA										
32.	CILIC, Marin CRO	[7]	R. NADAL [4]	6-1 1 6-4 1 6-2							
33.	NADAL, Rafael ESP	[4]									
34.	ISTOMIN, Denis UZB		A. KUZNETSOV	6-4 1 3-6 1 6-1 7-6(6)							
35.	ROBERT, Stephane FRA		A. RAMOS-VINOLAS [31]	7-5 1 6-4 1 7-6(5)							
36.	SEPPI, Andreas ITA										
37.	BELLUCCI, Thomas BRA		L. POUILLE [24]	6-1 1 6-4 1 6-2							
38.	KUZNETSOV, Andrey RUS										
39.	BENNETT, Julien FRA										
40.	RAMOS-VINOLAS, Albert ESP	[31]	L. POUILLE [24]	6-1 1 6-4 1 6-2							
41.	POUILLE, Lucas FRA	[24]									
42.	KUKUSHKIN, Mikhail KAZ		M. CHILJONELLI	6-1 1 6-4 1 6-2							
43.	CLUZAR, Guillaume BRA										
44.	CHILJONELLI, Marco SUI		F. DELBONIS	3-6 1 6-7(6) 1 6-2 1 6-4							
45.	BAKER, Brian USA		R. BAUTISTA AGUT [15]	5-7 1 6-2 1 6-3 1 6-2							
46.	DELBONIS, Federico ARG										
47.	GARCIA-LOPEZ, Guillermo ESP										
48.	BAUTISTA AGUT, Roberto ESP	[15]	G. MONFILS [10]	6-4 1 6-2 1 7-6(5)							
49.	MONFILS, Gael FRA	[10]									
50.	MULLER, Gilles LUX	(Q)	J. SATRAL	7-5 1 6-7(10) 1 7-6(3)							
51.	SATRAL, Jan CZE	(Q)									
52.	MCDONALD, Mackenzie USA	(W)	N. ALMAGRO	6-1 1 6-4 1 7-6(7)							
53.	FUCSOVICS, Marton HUN	(Q)	P. CUEVAS [18]	7-6(5) 1 6-4 1 7-6(9)							
54.	ALMAGRO, Nicolas ESP										
55.	SELA, Dudu ISR	[18]	M. BAGHDATIS	6-2 1 6-4 1 6-3 1 6-4							
56.	CUEVAS, Pablo URU	[32]									
57.	PAIRE, Benoit FRA										
58.	LACKO, Dustin SRB		R. HARRISON	6-4 1 6-2 1 1-1 Ret.							
59.	BAGHDATIS, Marcos CYP										
60.	BAGNIS, Facundo ARG										
61.	MANNARINO, Adrian FRA										
62.	HARRISON, Ryan USA	(Q)	M. RAONIC [5]	7-5 1 6-3 1 6-4							
63.	BROWN, Dustin GER										
64.	RAONIC, Milos CAN	[5]									

Champion:
WAWRINKA, Stan SUI
6-7(1) 1 6-4 1 7-5 1 6-3

An Introduction to the Introduction



US Open 2016
Men's Singles



Round 1	Round 2	Round 3	Round 4	Quarterfinals	Semifinals
65. THIEM, Dominic AUT 66. MULLMAN, John AUS 67. BERANKIS, Ricardas LTU 68. JAZIRI, Malek TUN 69. CARRRENO BUSTA, Pablo ESP 70. WASHKA, Ilya BLR 71. TIPSAREVIC, Janko SRB 72. QUERREY, Sam USA 73. JOHNSON, Steve USA 74. DONSKOY, Evgeniy RUS 75. DEL POTRO, Juan Martin ARG 76. SCHWARTZMAN, Diego ARG 77. GABASHVILI, Teymuraz RUS 78. FOGNINI, Fabio ITA 79. DOLGOPOLOV, Alexandr UKR 80. FERRER, David ESP 81. KYRGIOS, Nick AUS 82. BEDENE, Aljaz GBR 83. ZEBALLOS, Horacio ARG 84. MAYER, Florian GER 85. DODIG, Ivan CRO 86. MARCHENKO, Ilya UKR 87. DZUMHUR, Damir BLR 88. TOMIC, Bernard AUS 89. ZVEREV, Alexander GER 90. BRANDS, Daniel GER 91. EVANS, Daniel GBR 92. RAM, Ramesh USA 93. KUDLA, Denis USA 94. GIANNESSI, Alessandro ITA 95. VERDASCO, Fernando ESP 96. WAWRINKA, Stan SUI 97. NISHIKORI, Kei JPN 98. BECKER, Benjamin GER 99. KHACHANOV, Karen RUS 100. FABBIANO, Thomas ITA 101. HARRISON, Christian USA 102. MATHEU, Paul-Henri FRA 103. MAHUT, Nicolas FRA 104. KOHLSCHEIDER, Philipp GER 105. KARLOVIC, Ivo CRO 106. LI, Yen-Hsun TPE 107. YOUNG, Donald USA 108. STRUFF, Jan-Lennard GER 109. TRICKI, Viktor SRB 110. ALBOT, Radu MDA 111. DONALDSON, Jared USA 112. GOFFIN, David BEL 113. LOPEZ, Feliciano ESP 114. OSORIO, Boris CRO 115. ESTRELLA BURGOS, Victor DOM 116. SOUSA, Joao POR 117. MMOH, Michael AUS 118. CHARDY, Jeremy FRA 119. CERVANTES, Inigo ESP 120. DIMITROV, Grigor BUL 121. SMON, Gilles FRA 122. STEPANEK, Radek CZE 123. BERLOCCO, Carlos ARG 124. LORENZI, Paolo ITA 125. GRANOLLERS, Marc ESP 126. MONACO, Juan ARG 127. ROSOL, Lukas CZE 128. MURRAY, Andy GBR	[8] D. THIEM [8] 6-3 2-6 5-7 6-4 6-3 [8] B. BERANKIS 3-6 7-6(3) 6-4 1-6-2 [Q] P. CARRRENO BUSTA 6-0 7-5 1-6-2 [Q] J. TIPSAREVIC 7-6(4) 6-7(0) 6-3 6-3 [19] S. JOHNSON [19] 6-4 6-1 7-6(2) 1-6-3 6-3 (W) J. DEL POTRO 6-4 6-4 7-6(3) [E] EDGON 6-7(5) 3-6 7-6(5) 7-6 6-4 [D] FERRER [11] 6-5 Ret. [11] N. KYRGIOS [14] 6-4 6-4 1-6-4 [H] ZEBALLOS 6-3 6-4 1-6-2 [J] MARCHENKO 6-3 6-4 1-6-2 7-5 [D] DZUMHUR 6-4 6-3 4-6 7-6(0) [17] A. ZVEREV [27] (L) D. EVANS 3-6 6-1 6-4 7-6(4) (W) A. GIANNESSI 6-2 4-6 7-5 6-1 [Q] H. VERDASCO 0-6 6-4 6-1 1-6 6-0 [Q] S. WAWRINKA [3] 7-6(4) 6-4 1-6-4 [6] K. NISHIKORI [6] 6-1 6-1 3-6 1-6-3 [Q] K. KHACHANOV 6-4 4-6 6-4 6-3 [Q] P. MATHEU 6-0 6-2 1-6-1 [N] MAHUT 6-4 6-4 6-2 [25] K. NISHIKORI [6] 6-3 7-5 1-6 Ret. [21] I. KARLOVIC [21] 4-6 7-6(4) 1-6-7(4) 7-6(5) 1-7-5 [D] YOUNG 6-3 7-5 1-6-4 7-5 [V] TRICKI 6-7 3-6 6-4 6-4 7-6(3) [J] DONALDSON 7-5 6-3 6-3 [Q] J. DONALDSON 4-6 7-5 6-4 6-0 [12] J. GOFFIN 4-6 7-5 6-4 6-0 [16] F. LOPEZ [16] 3-4 Ret. [J] SOUSA 6-2 6-4 1-6 1-6 7-5 (W) G. DIMITROV [22] 6-0 6-1 6-1 [G] DIMITROV 6-4 6-4 1-6-1 [G] DIMITROV [22] 4-6 6-4 1-6 6-1 6-4 6-2 [22] G. DIMITROV 6-2 6-4 1-6-1 [30] A. SMON 6-3 6-1 6-4 [Q] P. LORENZI 3-6 6-2 6-2 6-2 6-7(1) 7-6(3) 6-4 6-2 1-6-1 [M] GRANOLLERS 7-6(5) 7-6(2) 6-4 [A] MURRAY [2] 6-3 6-2 6-2	[8] D. THIEM [8] 6-4 6-3 6-2 [P. CARRRENO BUSTA] 3-6 4-6 6-1 6-4 6-4 [J. TIPSAREVIC] 3-6 4-6 6-1 6-4 6-4 [J. DEL POTRO] 7-6(5) 1-6-3 6-2 [J. DEL POTRO] 7-6(3) 1-6-2 6-3 [D. FERRER] 6-0 4-6 5-7 6-1 6-4 [N. KYRGIOS] 7-5 6-4 1-6-4 [I. MARCHENKO] 4-6 6-4 1-6-1 Ret. [I. MARCHENKO] 6-3 6-4 1-6-2 7-5 [D. DZUMHUR] 6-4 6-3 4-6 7-6(0) [D. EVANS] 6-4 6-4 5-7 6-2 [A. GIANNESSI] 4-6 6-3 6-7(6) 7-6(8) 1-6-2 [S. WAWRINKA] 4-6 6-3 6-7(6) 7-6(8) 1-6-2 [K. NISHIKORI] 6-4 4-6 6-4 6-3 [K. NISHIKORI] 4-6 6-1 6-2 1-6-2 [K. NISHIKORI] 6-3 7-5 1-6 Ret. [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [J. DONALDSON] 7-5 6-3 6-3 [J. DONALDSON] 4-6 7-5 6-4 6-0 [J. GOFFIN] 4-6 7-5 6-4 6-0 [F. LOPEZ] 3-4 Ret. [J. SOUSA] 6-2 6-4 1-6 1-6 7-5 [G. DIMITROV] 6-4 6-4 1-6-1 [G. DIMITROV] 4-6 6-4 1-6 6-1 6-4 6-2 [G. DIMITROV] 6-2 6-4 1-6-1 [A. SMON] 6-3 6-1 6-4 [P. LORENZI] 3-6 6-2 6-2 6-2 6-7(1) 7-6(3) [M. GRANOLLERS] 7-6(5) 7-6(2) 6-4 [A. MURRAY] 6-3 6-2 6-2	[J. DEL POTRO] 6-3 3-2 Ret. [J. DEL POTRO] 7-6(3) 1-6-2 6-3 [S. WAWRINKA] 7-6(5) 4-6 6-3 1-6-2 [I. MARCHENKO] 4-6 6-4 1-6-1 Ret. [S. WAWRINKA] 6-4 6-1 6-7(5) 1-6-3 [S. WAWRINKA] 4-6 6-3 6-7(6) 7-6(8) 1-6-2 [S. WAWRINKA] 4-6 7-5 1-6 1-6-2 [K. NISHIKORI] 4-6 6-1 6-2 1-6-2 [K. NISHIKORI] 6-3 7-5 1-6 Ret. [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [J. DONALDSON] 7-5 6-3 6-3 [J. DONALDSON] 4-6 7-5 6-4 6-0 [J. GOFFIN] 4-6 7-5 6-4 6-0 [F. LOPEZ] 3-4 Ret. [J. SOUSA] 6-2 6-4 1-6 1-6 7-5 [G. DIMITROV] 6-4 6-4 1-6-1 [G. DIMITROV] 4-6 6-4 1-6 6-1 6-4 6-2 [G. DIMITROV] 6-2 6-4 1-6-1 [A. SMON] 6-3 6-1 6-4 [P. LORENZI] 3-6 6-2 6-2 6-2 6-7(1) 7-6(3) [M. GRANOLLERS] 7-6(5) 7-6(2) 6-4 [A. MURRAY] 6-3 6-2 6-2	[S. WAWRINKA] 7-6(5) 4-6 6-3 1-6-2 [S. WAWRINKA] 6-4 6-1 6-7(5) 1-6-3 [S. WAWRINKA] 4-6 7-5 1-6 1-6-2 [K. NISHIKORI] 4-6 6-1 6-2 1-6-2 [K. NISHIKORI] 6-3 7-5 1-6 Ret. [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [J. DONALDSON] 7-5 6-3 6-3 [J. DONALDSON] 4-6 7-5 6-4 6-0 [J. GOFFIN] 4-6 7-5 6-4 6-0 [F. LOPEZ] 3-4 Ret. [J. SOUSA] 6-2 6-4 1-6 1-6 7-5 [G. DIMITROV] 6-4 6-4 1-6-1 [G. DIMITROV] 4-6 6-4 1-6 6-1 6-4 6-2 [G. DIMITROV] 6-2 6-4 1-6-1 [A. SMON] 6-3 6-1 6-4 [P. LORENZI] 3-6 6-2 6-2 6-2 6-7(1) 7-6(3) [M. GRANOLLERS] 7-6(5) 7-6(2) 6-4 [A. MURRAY] 6-3 6-2 6-2	[S. WAWRINKA] 7-6(5) 4-6 6-3 1-6-2 [S. WAWRINKA] 6-4 6-1 6-7(5) 1-6-3 [S. WAWRINKA] 4-6 7-5 1-6 1-6-2 [K. NISHIKORI] 4-6 6-1 6-2 1-6-2 [K. NISHIKORI] 6-3 7-5 1-6 Ret. [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [I. KARLOVIC] 6-4 7-6(4) 1-6-4 [J. DONALDSON] 7-5 6-3 6-3 [J. DONALDSON] 4-6 7-5 6-4 6-0 [J. GOFFIN] 4-6 7-5 6-4 6-0 [F. LOPEZ] 3-4 Ret. [J. SOUSA] 6-2 6-4 1-6 1-6 7-5 [G. DIMITROV] 6-4 6-4 1-6-1 [G. DIMITROV] 4-6 6-4 1-6 6-1 6-4 6-2 [G. DIMITROV] 6-2 6-4 1-6-1 [A. SMON] 6-3 6-1 6-4 [P. LORENZI] 3-6 6-2 6-2 6-2 6-7(1) 7-6(3) [M. GRANOLLERS] 7-6(5) 7-6(2) 6-4 [A. MURRAY] 6-3 6-2 6-2

An Introduction to the Introduction

Champion:
WAWRINKA, Stan SUI
6-7(1) | 6-4 | 7-5 | 6-3

US Open 2016
Men's Singles

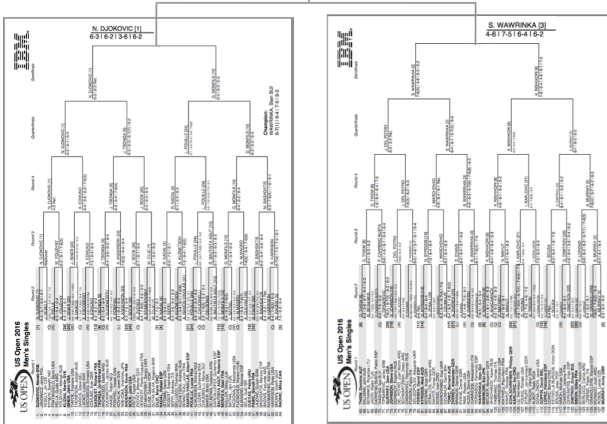
US OPEN
Men's Singles

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841

[illegible]

An Introduction to the Introduction

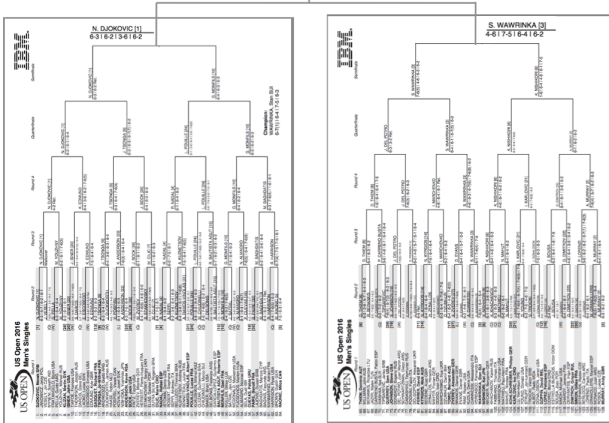
Champion:
WAWRINKA, Stan SUI
6-7(1) | 6-4 | 7-5 | 6-3



- 128 players in total;

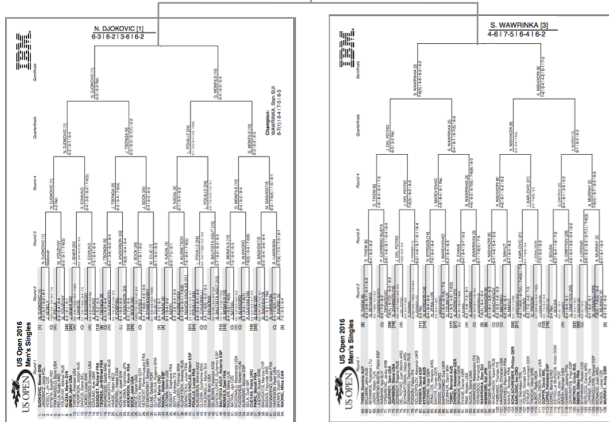
An Introduction to the Introduction

Champion:
WAWRINKA, Stan SUI
6-7(1) | 6-4 | 7-5 | 6-3



- 128 players in total;
- 127 matches were played to determine who was the champion.

An Introduction to the Introduction



- 128 players in total;
- 127 matches were played to determine who was the champion.
- **Is it possible to just play fewer matches?**

An Introduction to the Introduction



An Introduction to the Introduction



You were called to answer this question;

An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;

An Introduction to the Introduction

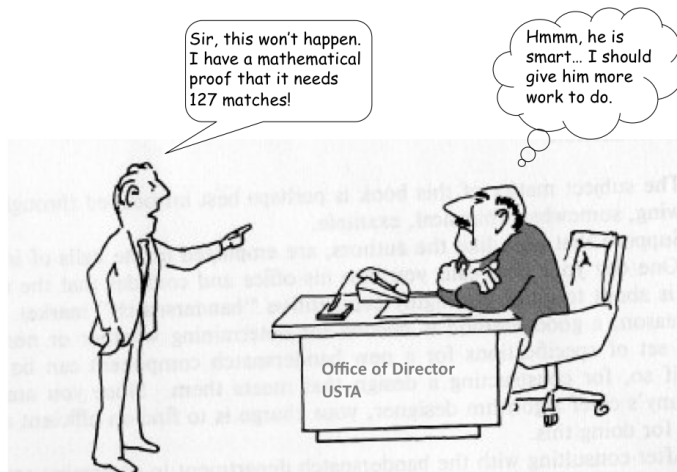
- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were **necessary**,

An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were **necessary**, and finally came up with a mathematical proof for that.

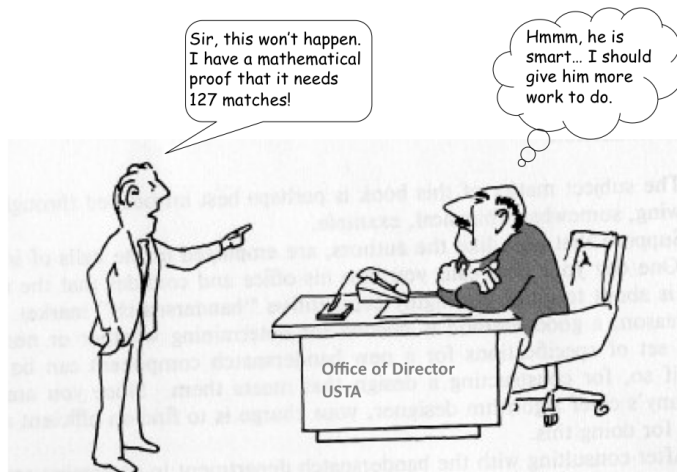
An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were **necessary**, and finally came up with a mathematical proof for that.



An Introduction to the Introduction

- You tried various match formats, but all need at least 127 matches;
- Then you suspected that 127 matches were **necessary**, and finally came up with a mathematical proof for that.



An Introduction to the Introduction

Same problems

TENNIS TOURNAMENT

FINDING MAXIMUM

Input:

128 players

n numbers

Output:

Champion

the maximum number

Solution:

a match scheme

an algorithm

Time Efficiency:

number of matches

number of comparisons

An Introduction to the Introduction

Same problems

TENNIS TOURNAMENT

FINDING MAXIMUM

Input:

128 players

n numbers

Output:

Champion

the maximum number

Solution:

a match scheme

an algorithm

Time Efficiency:

number of matches

number of comparisons

You actually accomplished two tasks:

An Introduction to the Introduction

Same problems	TENNIS TOURNAMENT	FINDING MAXIMUM
Input:	128 players	n numbers
Output:	Champion	the maximum number
Solution:	a match scheme	an algorithm
Time Efficiency:	number of matches	number of comparisons

You actually accomplished two tasks:

- **tried various algorithms** for TENNIS TOURNAMENT before giving up;

An Introduction to the Introduction

Same problems	TENNIS TOURNAMENT	FINDING MAXIMUM
Input:	128 players	n numbers
Output:	Champion	the maximum number
Solution:	a match scheme	an algorithm
Time Efficiency:	number of matches	number of comparisons

You actually accomplished two tasks:

- **tried various algorithms** for TENNIS TOURNAMENT before giving up;
- **proved that more efficient algorithms do not exist**;

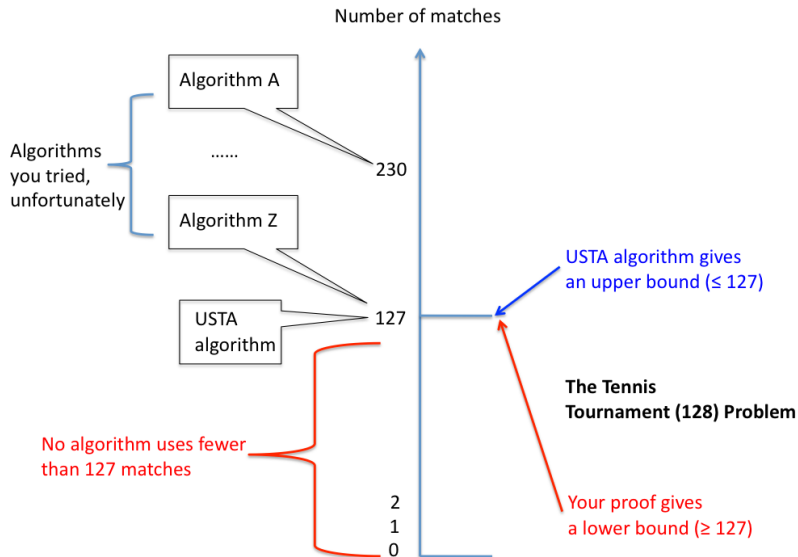
An Introduction to the Introduction

Same problems	TENNIS TOURNAMENT	FINDING MAXIMUM
Input:	128 players	n numbers
Output:	Champion	the maximum number
Solution:	a match scheme	an algorithm
Time Efficiency:	number of matches	number of comparisons

You actually accomplished two tasks:

- **tried various algorithms** for TENNIS TOURNAMENT before giving up;
- **proved that more efficient algorithms do not exist**;
- actually you proved the “USTA algorithm” was already the optimal.

An Introduction to the Introduction



An Introduction to the Introduction

In general, for a given problem Π ,

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T ,

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T , gives a **lower bound** T for Π ;

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T , gives a **lower bound** T for Π ;
a proof that Π cannot have algorithms faster than time S , with $S > T$,

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T , gives a **lower bound** T for Π ;
a proof that Π cannot have algorithms faster than time S , with $S > T$, gives a **tighter lower bound** S for Π .

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π ,
where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a
tighter upper bound T_B for Π ;
- a proof that Π cannot have algorithms faster than time T ,
gives a **lower bound** T for Π ;
a proof that Π cannot have algorithms faster than time S , with $S > T$,
gives a **tighter lower bound** S for Π .
- for problem Π , time **lower bounds** \leq time **upper bounds**;

An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T , gives a **lower bound** T for Π ;
a proof that Π cannot have algorithms faster than time S , with $S > T$, gives a **tighter lower bound** S for Π .
- for problem Π , time **lower bounds** \leq time **upper bounds**;
when a **lower bound** is the same as an **upper bound**, both the lower and upper bounds are called **optimal**.

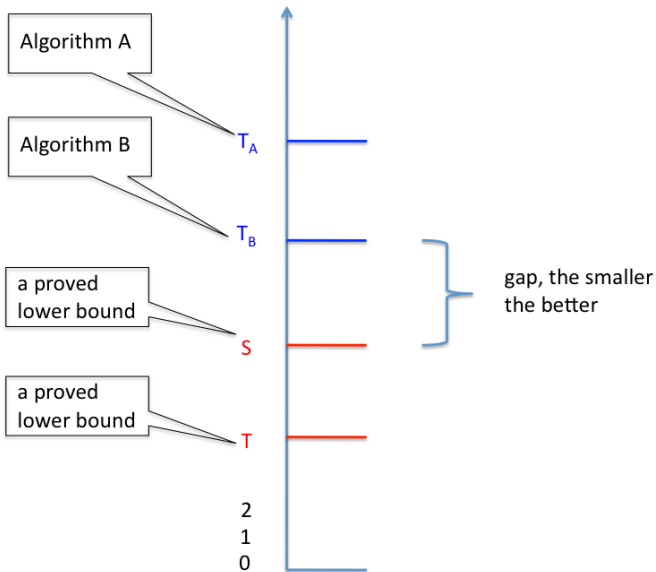
An Introduction to the Introduction

In general, for a given problem Π ,

- an algorithm A for Π also gives a time **upper bound** T_A for Π , where T_A is the time complexity of algorithm A ;
a faster algorithm B for Π , with $T_B < T_A$, gives a **tighter upper bound** T_B for Π ;
- a proof that Π cannot have algorithms faster than time T , gives a **lower bound** T for Π ;
a proof that Π cannot have algorithms faster than time S , with $S > T$, gives a **tighter lower bound** S for Π .
- for problem Π , time **lower bounds** \leq time **upper bounds**;
when a **lower bound** is the same as an **upper bound**, both the lower and upper bounds are called **optimal**.
Also the corresponding algorithms (that offer the upper bound) are called **optimal** for Π .

An Introduction to the Introduction

Time complexity situation for problem Π



An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds

An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds

- we need be familiar with techniques for algorithm complexity analysis;

An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds

- we need be familiar with techniques for algorithm complexity analysis;
- we need to master efficient algorithm design skills;

An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds

- we need be familiar with techniques for algorithm complexity analysis;
- we need to master efficient algorithm design skills;

2. and to achieve tighter lower bounds,

An Introduction to the Introduction

So to design good algorithms for computational problems, our goals are

1. to achieve tighter upper bounds

- we need be familiar with techniques for algorithm complexity analysis;
- we need to master efficient algorithm design skills;

2. and to achieve tighter lower bounds,

- we need to know the methods for lower bound proofs.

The Introduction

The Introduction

- ▶ What is this course about (and why is it needed)?

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)
- ▶ Goals to achieve

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)
- ▶ Goals to achieve
 - to learn to measure performance of algorithms

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)
- ▶ Goals to achieve
 - to learn to measure performance of algorithms
 - to master some fundamental algorithmic techniques

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)
- ▶ Goals to achieve
 - to learn to measure performance of algorithms
 - to master some fundamental algorithmic techniques
 - to study advanced algorithmic skills

The Introduction

- ▶ What is this course about (and why is it needed)?
 - about basic yet indispensable skills for problem solving
 - algorithm design leads to writing code,
i.e., creative thinking without programming languages
- ▶ How different is this course from other algorithm courses?
 - design technique-oriented, not application-oriented
 - emphasis on guaranteed performance (typically in efficiency)
- ▶ Goals to achieve
 - to learn to measure performance of algorithms
 - to master some fundamental algorithmic techniques
 - to study advanced algorithmic skills
 - to understand computational intractability

Part I. Foundations

Part I. Foundations

- ▶ Chapter 1. The role of algorithms in computing
- ▶ Chapter 2. Getting started
- ▶ Chapter 3. Growth of functions
- ▶ Chapter 4. Solving recurrences
- ▶ Chapter 5. Probabilistic analysis and randomized algorithms

Part I. Foundations

The theme of the course

Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms

Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms
- mean: through developing skills to analyze algorithms

Part I. Foundations

The theme of the course

- Goal: learning techniques to design efficient algorithms
- mean: through developing skills to analyze algorithms

Design and analysis of algorithms are closely related.

Part I. Foundations

Example: the Fibonacci sequence.

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n \geq 3 \\ 1, & \text{otherwise} \end{cases}$$

Part I. Foundations

Example: the Fibonacci sequence.

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n \geq 3 \\ 1, & \text{otherwise} \end{cases}$$

That is:

n	:	1	2	3	4	5	6	7	8	9	...
$f(n)$:	1	1	2	3	5	8	13	21	34	...

Part I. Foundations

Problem 1: Computing the n th Fibonacci number:

Part I. Foundations

Problem 1: Computing the n th Fibonacci number:

INPUT: $n \geq 1$;

OUTPUT: the n th number in the Fibonacci sequence.

Part I. Foundations

Problem 1: Computing the n th Fibonacci number:

INPUT: $n \geq 1$;

OUTPUT: the n th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*

Part I. Foundations

Problem 1: Computing the n th Fibonacci number:

INPUT: $n \geq 1$;

OUTPUT: the n th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*

- recursive: task decomposition, top-down, recursive calls;

Part I. Foundations

Problem 1: Computing the n th Fibonacci number:

INPUT: $n \geq 1$;

OUTPUT: the n th number in the Fibonacci sequence.

Two different types of algorithms: *recursive* and *iterative*

- recursive: task decomposition, top-down, recursive calls;
- iterative: more tightly coupled tasks, bottom-up approaches;

Part I. Foundations

REC-FIBONACCI(n)

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ **or** $n = 2$,

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ **or** $n = 2$, **return** (1);

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ or $n = 2$, **return** (1);

else

$T_1 = \text{REC-FIBONACCI}(n - 1);$

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ or $n = 2$, **return** (1);

else

$T_1 = \text{REC-FIBONACCI}(n - 1);$

$T_2 = \text{REC-FIBONACCI}(n - 2);$

Part I. Foundations

REC-FIBONACCI(n)

if $n = 1$ or $n = 2$, **return** (1);

else

$T_1 = \text{REC-FIBONACCI}(n - 1);$

$T_2 = \text{REC-FIBONACCI}(n - 2);$

return ($T_1 + T_2$);

Part I. Foundations

REC-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$ , return (1);  
else  
     $T_1 = \text{REC-FIBONACCI}(n - 1);$   
     $T_2 = \text{REC-FIBONACCI}(n - 2);$   
    return ( $T_1 + T_2$ );
```

But how efficient is it? Or how slow is it?

Part I. Foundations

REC-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$ , return (1);  
else  
     $T_1 = \text{REC-FIBONACCI}(n - 1);$   
     $T_2 = \text{REC-FIBONACCI}(n - 2);$   
    return ( $T_1 + T_2$ );
```

But how efficient is it? Or how slow is it?

its execution is via a *run-time stack*

Part I. Foundations

REC-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$ , return (1);  
else  
     $T_1 = \text{REC-FIBONACCI}(n - 1);$   
     $T_2 = \text{REC-FIBONACCI}(n - 2);$   
    return ( $T_1 + T_2$ );
```

But how efficient is it? Or how slow is it?

its execution is via a *run-time stack*

- suitable for execution of subroutines

Part I. Foundations

REC-FIBONACCI(n)

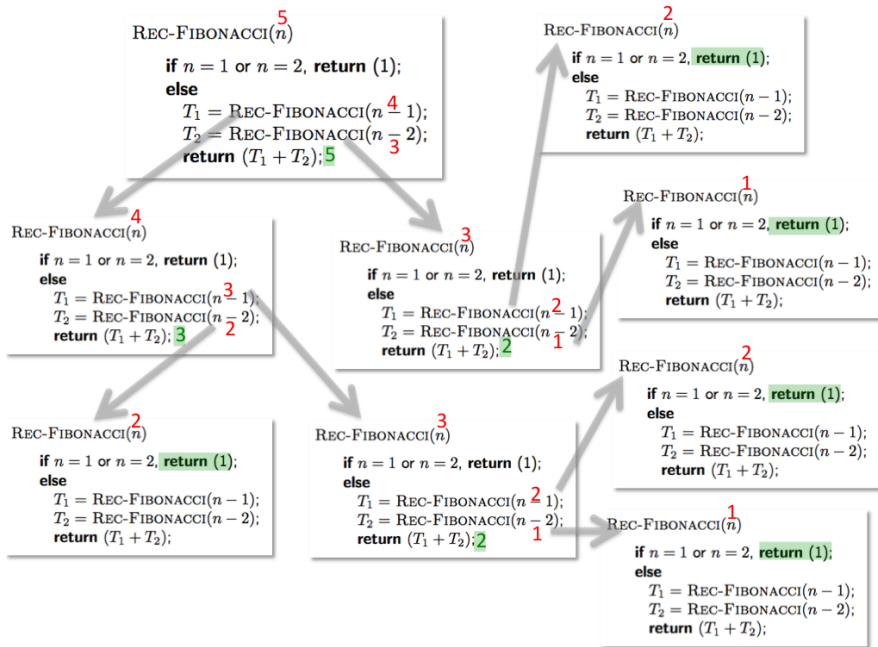
```
if  $n = 1$  or  $n = 2$ , return (1);  
else  
     $T_1 = \text{REC-FIBONACCI}(n - 1);$   
     $T_2 = \text{REC-FIBONACCI}(n - 2);$   
    return ( $T_1 + T_2$ );
```

But how efficient is it? Or how slow is it?

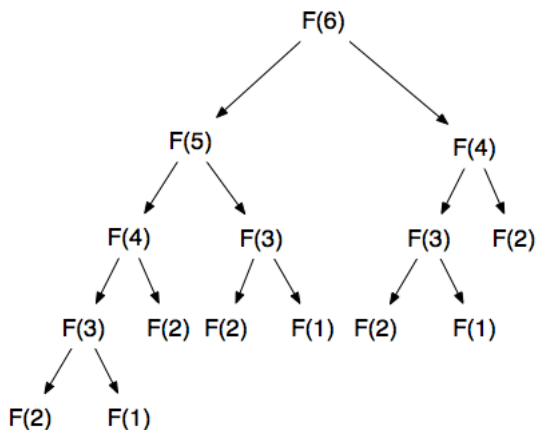
its execution is via a *run-time stack*

- suitable for execution of subroutines
- but oblivious, cannot remember any completed subroutine.

Part I. Foundations

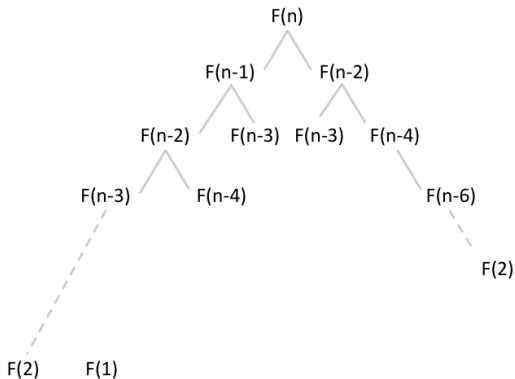


Part I. Foundations



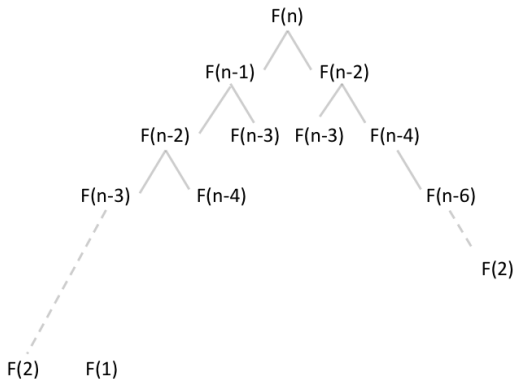
Repeated computations everywhere!

Part I. Foundations



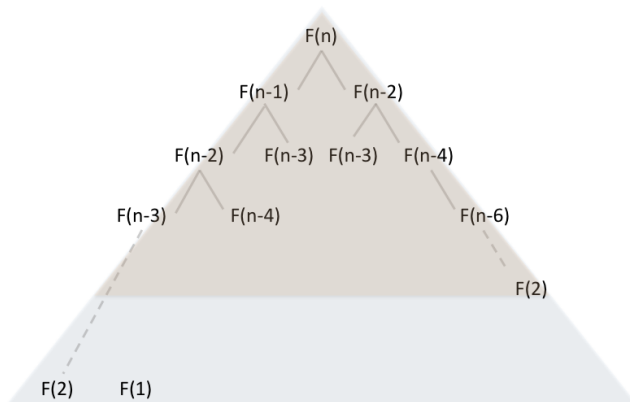
The size of tree is the number of recursive calls;

Part I. Foundations

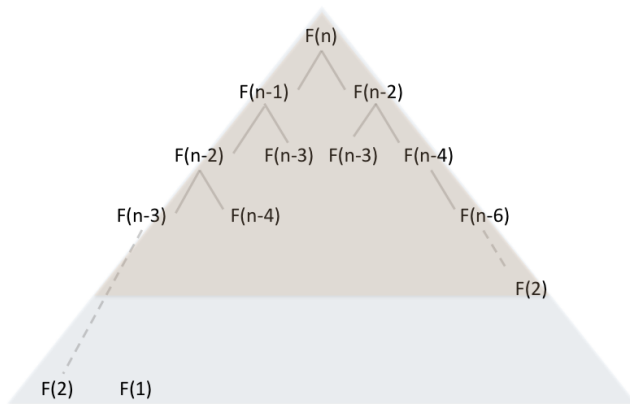


The size of tree is the number of recursive calls;
How big is it?

Part I. Foundations

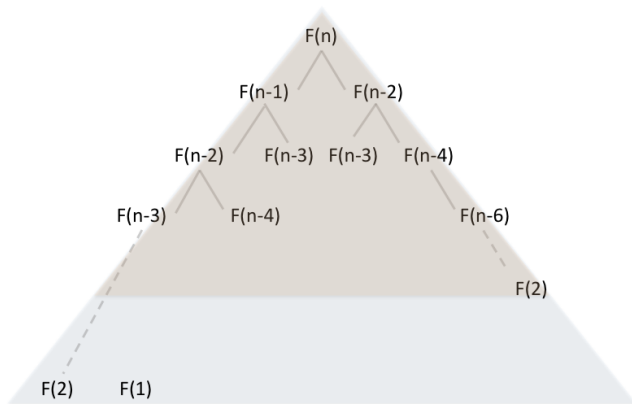


Part I. Foundations



small triangle \leq size of tree \leq large triangle

Part I. Foundations



small triangle \leq size of tree \leq large triangle

roughly: $2^{\frac{n}{2}} \leq$ size of tree $\leq 2^n$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

return ($M[n]$)

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

return ($M[n]$)

How fast is it?

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ **or** $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$,

Part I. Foundations

ITERATIVE-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$  return (1);  
else  
     $M[1] = 1, M[2] = 1$   
    for  $i = 3$  to  $n$  do  
         $M[i] = M[i - 1] + M[i - 2]$   
    return ( $M[n]$ )
```

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for}$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$  return (1);  
else  
     $M[1] = 1, M[2] = 1$   
    for  $i = 3$  to  $n$  do  
         $M[i] = M[i - 1] + M[i - 2]$   
    return ( $M[n]$ )
```

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

if $n = 1$ or $n = 2$ **return** (1);

else

$M[1] = 1, M[2] = 1$

for $i = 3$ **to** n **do**

$M[i] = M[i - 1] + M[i - 2]$

return ($M[n]$)

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$$T_{total} \leq c_1$$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$  return (1);  
else  
     $M[1] = 1, M[2] = 1$   
    for  $i = 3$  to  $n$  do  
         $M[i] = M[i - 1] + M[i - 2]$   
    return ( $M[n]$ )
```

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$$T_{total} \leq c_1 + c_2 + d(n - 2),$$

Part I. Foundations

ITERATIVE-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$  return (1);  
else  
     $M[1] = 1, M[2] = 1$   
    for  $i = 3$  to  $n$  do  
         $M[i] = M[i - 1] + M[i - 2]$   
    return ( $M[n]$ )
```

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$T_{total} \leq c_1 + c_2 + d(n - 2)$, a linear function in n

Part I. Foundations

ITERATIVE-FIBONACCI(n)

```
if  $n = 1$  or  $n = 2$  return (1);  
else  
     $M[1] = 1, M[2] = 1$   
    for  $i = 3$  to  $n$  do  
         $M[i] = M[i - 1] + M[i - 2]$   
    return ( $M[n]$ )
```

How fast is it?

$$T_{total} = \max\{T_{if}, T_{else}\}$$

where $T_{if} = c_1$, $T_{else} = c_2 + T_{for} = c_2 + d \times (n - 2)$

$T_{total} \leq c_1 + c_2 + d(n - 2)$, a linear function in n

ITERATIVE-FIBONACCI(n) is a simple **dynamic programming algorithm**.

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

To fill out a table of size n , with

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

To fill out a table of size n , with

- each entry being filled out exactly once, and

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

To fill out a table of size n , with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say c steps.

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

To fill out a table of size n , with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say c steps.

So the total time `ITERATIVE-FIBONACCI(n)` uses is

Part I. Foundations

Actually, all the algorithm `ITERATIVE-FIBONACCI(n)` does is:

To fill out a table of size n , with

- each entry being filled out exactly once, and
- filling out an entry takes a constant, say c steps.

So the total time `ITERATIVE-FIBONACCI(n)` uses is

$$T(n) = c \times n$$

Chapter 1. The Role of Algorithms in Computing

Chapter 1. The role of algorithms in computing

Chapter 1. The Role of Algorithms in Computing

Chapter 1. The role of algorithms in computing

What is an Algorithm: a well-defined, finite procedure that takes an input and produces an output.

Chapter 1. The Role of Algorithms in Computing

Chapter 1. The role of algorithms in computing

What is an Algorithm: a well-defined, finite procedure that takes an input and produces an output.

Example 2: An algorithm skeleton;

Algorithm MAXIMUM;

INPUT: list $X = \{a_1, \dots, a_n\}$;

Body that is a series of instructions;

OUTPUT: y , the maximum of a_1, \dots, a_n .

Chapter 1. The Role of Algorithms in Computing

Alternatively, an algorithm specifies a finite process to **compute a function or a relation**.

Chapter 1. The Role of Algorithms in Computing

Alternatively, an algorithm specifies a finite process to **compute a function or a relation**.

e.g., algorithm MAXIMUM computes the following function:

$$f_{\max}(X) = y, \text{ where } \forall a \in X, y \geq a,$$

.

Chapter 1. The Role of Algorithms in Computing

Alternatively, an algorithm specifies a finite process to **compute a function or a relation**.

e.g., algorithm MAXIMUM computes the following function:

$$f_{\max}(X) = y, \text{ where } \forall a \in X, y \geq a,$$

.

For some problems, the functions computed are **predicates**, i.e., output $y \in \{\text{TRUE}, \text{FALSE}\}$

Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:

Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:

- very large input data for “easy” problems;

Chapter 1. The Role of Algorithms in Computing

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space is necessary.

Two typical situations:

- very large input data for “easy” problems;
- moderately large input data for “hard” problems.

Chapter 2. Getting Started

Chapter 2. Getting started

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that

$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Insertion Sort

idea: an iterative process to produce a new list such that

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Insertion Sort

idea: an iterative process to produce a new list such that
at each iteration, the new list consists of two sublists,

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Insertion Sort

idea: an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,

- a **sorted sublist** followed by an **unsorted sublist**, and

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Insertion Sort

idea: an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,

- a **sorted sublist** followed by an **unsorted sublist**, and
- the leftmost number of the **unsorted** is being inserted into the **sorted**.

Chapter 2. Getting Started

Chapter 2. Getting started

The Sorting Problem

INPUT: n numbers $\langle a_1, \dots, a_n \rangle$;

OUTPUT: a reordering $\langle a'_1, \dots, a'_n \rangle$ of the input such that
$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

Insertion Sort

idea: an iterative process to produce a new list such that

at each iteration, the new list consists of two sublists,

- a **sorted sublist** followed by an **unsorted sublist**, and
- the leftmost number of the **unsorted** is being inserted into the **sorted**.

As the process goes, the **sorted sublist** gets longer, the **unsorted sublist** gets shorter, until the **unsorted** becomes empty.

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

1 **for** $j = 2$ **to** $length[A]$ **do**

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $length[A]$  do  
2     $key = A[j]$ 
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j-1]$ }  
4     $i = j - 1$ 
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }  
4     $i = j - 1$   
5    while  $i > 0$  and  $A[i] > \text{key}$ 
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }  
4     $i = j - 1$   
5    while  $i > 0$  and  $A[i] > \text{key}$   
6      do  $A[i + 1] = A[i]$ 
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }  
4     $i = j - 1$   
5    while  $i > 0$  and  $A[i] > \text{key}$   
6      do  $A[i + 1] = A[i]$   
7       $i = i - 1$ 
```


Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do  
2     $\text{key} = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }  
4     $i = j - 1$   
5    while  $i > 0$  and  $A[i] > \text{key}$   
6      do  $A[i + 1] = A[i]$   
7         $i = i - 1$   
8     $A[i + 1] = \text{key}$ 
```

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $length[A]$  do  
2     $key = A[j]$   
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }  
4     $i = j - 1$   
5    while  $i > 0$  and  $A[i] > key$   
6      do  $A[i + 1] = A[i]$   
7         $i = i - 1$   
8     $A[i + 1] = key$ 
```

Analysis of the algorithm:

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $length[A]$  do
2     $key = A[j]$ 
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6      do  $A[i + 1] = A[i]$ 
7       $i = i - 1$ 
8     $A[i + 1] = key$ 
```

Analysis of the algorithm:

- (correctness proof): to show that the algorithm is as desired;

Chapter 2. Getting Started

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $length[A]$  do
2     $key = A[j]$ 
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6      do  $A[i + 1] = A[i]$ 
7         $i = i - 1$ 
8     $A[i + 1] = key$ 
```

Analysis of the algorithm:

- (correctness proof): to show that the algorithm is as desired;
- (efficiency proof): to show a guaranteed efficiency of the algorithm

Chapter 2. Getting Started

Correctness proof: this is to prove

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

If the algorithm consists of sequential blocks of instructions,
the task is to prove the correct transformation by each block.

Chapter 2. Getting Started

Correctness proof: this is to prove

the pre-condition (condition for the input)

is transformed by **the algorithm** to

the post-condition (condition for the output)

If the algorithm consists of sequential blocks of instructions,
the task is to prove the correct transformation by each block.

This means we need to prove that every sequential statement
in the algorithm transforms the given pre-condition to
the given post-condition.

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Finding **loop invariant** becomes necessary and sufficient.

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Finding **loop invariant** becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Finding **loop invariant** becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

at each iteration, the sublist $A[1..j-1]$ consists of the elements originally in the positions $[1..j-1]$ but in sorted order.

Chapter 2. Getting Started

The most difficult task is to do this for a loop statement.

Finding **loop invariant** becomes necessary and sufficient.

In INSERTION-SORT, the loop invariant is

at each iteration, the sublist $A[1..j-1]$ consists of the elements originally in the positions $[1..j-1]$ but in sorted order.

However, finding loop invariants is difficult!

Chapter 2. Getting Started

Efficiency analysis: This is to show that

Chapter 2. Getting Started

Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.

Chapter 2. Getting Started

Efficiency analysis: This is to show that

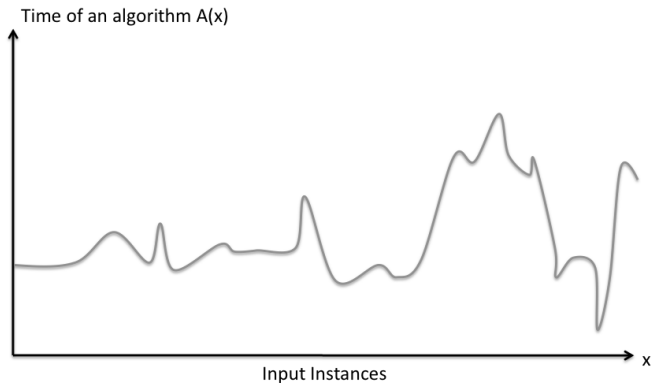
- For all cases of input, the needed computation resources for the algorithm.
- resources can be CPU time and memory space used in the computation.

Chapter 2. Getting Started

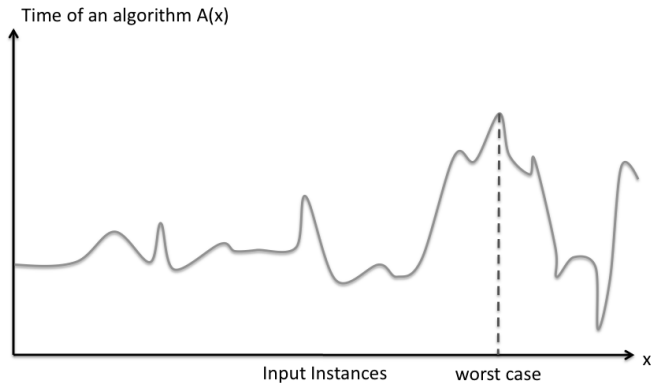
Efficiency analysis: This is to show that

- For all cases of input, the needed computation resources for the algorithm.
- resources can be CPU time and memory space used in the computation.
- however, the unit measured is not real time or memory unit.

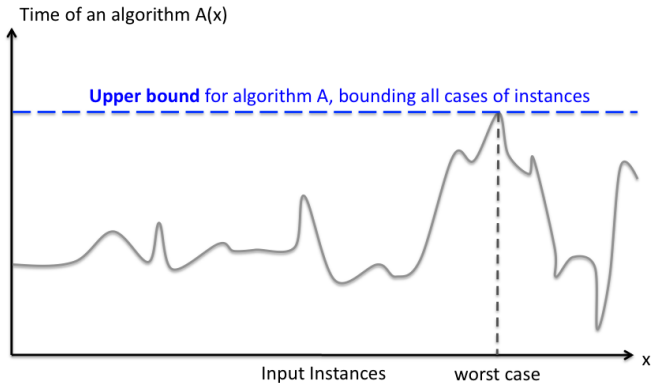
Chapter 2. Getting Started



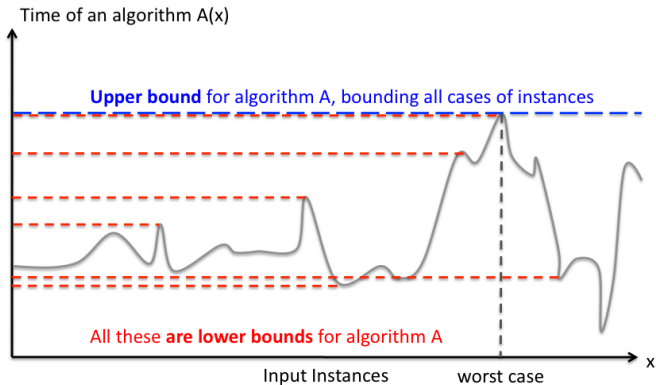
Chapter 2. Getting Started



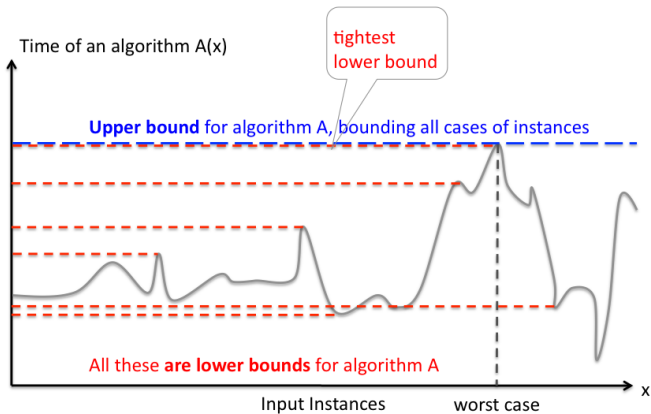
Chapter 2. Getting Started



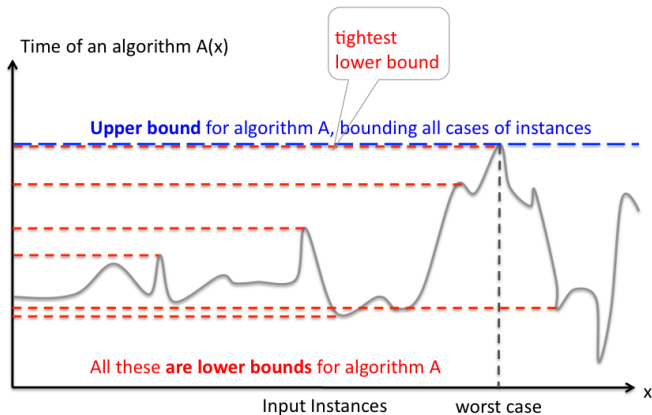
Chapter 2. Getting Started



Chapter 2. Getting Started



Chapter 2. Getting Started



lower bounds \leq worst case time \leq upper bounds

Chapter 2. Getting Started

Resource measurement based on

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
- counting primitive operations: addition, subtraction, floor, ceiling, multiplication, jump, memory movement,

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
- counting primitive operations: addition, subtraction, floor, ceiling, multiplication, jump, memory movement,
these operations differs in time by a constant multiplicative factor.

Chapter 2. Getting Started

Resource measurement based on

- random-access machine (RAM)
- counting primitive operations: addition, subtraction, floor, ceiling, multiplication, jump, memory movement,
these operations differs in time by a constant multiplicative factor.
- speed between different machines: a constant multiplicative factor.

Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $length[A]$  do
2     $key = A[j]$ 
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6      do  $A[i + 1] = A[i]$ 
7         $i = i - 1$ 
8     $A[i + 1] = key$ 
```


Chapter 2. Getting Started

Analysis of

Algorithm INSERTION-SORT(A)

```
1  for  $j = 2$  to  $\text{length}[A]$  do
2     $\text{key} = A[j]$ 
3    {Insert  $A[j]$  into sorted  $A[1..j - 1]$ }
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > \text{key}$ 
6      do  $A[i + 1] = A[i]$ 
7         $i = i - 1$ 
8     $A[i + 1] = \text{key}$ 
```

Assume t_j to be the number of times **while** is executed for every j .

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

$$T(n) \leq a \sum_{j=2}^n t_j + bn + c$$

for some constants a, b, c ,

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

$$T(n) \leq a \sum_{j=2}^n t_j + bn + c$$

for some constants a, b, c , for example, $a \geq c_5 + c_6 + c_7$, $b \geq c_1 + c_2 + c_4 + c_8$.

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

$$T(n) \leq a \sum_{j=2}^n t_j + bn + c$$

for some constants a, b, c , for example, $a \geq c_5 + c_6 + c_7$, $b \geq c_1 + c_2 + c_4 + c_8$.

Because $t_j = j$ in the worst case (e.g., list is reversely sorted).

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

$$T(n) \leq a \sum_{j=2}^n t_j + bn + c$$

for some constants a, b, c , for example, $a \geq c_5 + c_6 + c_7$, $b \geq c_1 + c_2 + c_4 + c_8$.

Because $t_j = j$ in the worst case (e.g., list is reversely sorted).

$$T(n) \leq a \frac{n}{2}(n+1) + bn + c - a$$

Chapter 2. Getting Started

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

$$T(n) \leq a \sum_{j=2}^n t_j + bn + c$$

for some constants a, b, c , for example, $a \geq c_5 + c_6 + c_7$, $b \geq c_1 + c_2 + c_4 + c_8$.

Because $t_j = j$ in the worst case (e.g., list is reversely sorted).

$$T(n) \leq a \frac{n}{2}(n+1) + bn + c - a \leq xn^2 + yn + z$$

for some constants x, y, z .

Chapter 2. Getting Started

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

I.e.,

$$un^2 + vn + w \leq T(n) \leq xn^2 + yn + z$$

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

I.e.,

$$un^2 + vn + w \leq T(n) \leq xn^2 + yn + z$$

\leq means **at least one case** for which \leq holds;

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

I.e.,

$$un^2 + vn + w \leq T(n) \leq xn^2 + yn + z$$

\leq means **at least one case** for which \leq holds;

$un^2 + vn + w$ is a complexity **lower bound** for $T(n)$

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

I.e.,

$$un^2 + vn + w \leq T(n) \leq xn^2 + yn + z$$

\leq means **at least one case** for which \leq holds;

$un^2 + vn + w$ is a complexity **lower bound** for $T(n)$

\leq means **in all cases** for which \leq holds;

Chapter 2. Getting Started

So we have proved:

$$T(n) \leq xn^2 + yn + z \quad \text{for some constants } x, y, z$$

We can also prove that (can you?)

$$T(n) \geq un^2 + vn + w \quad \text{for some constants } u, v, w$$

I.e.,

$$un^2 + vn + w \leq T(n) \leq xn^2 + yn + z$$

\geq means **at least one case** for which \geq holds;

$un^2 + vn + w$ is a complexity **lower bound** for $T(n)$

\leq means **in all cases** for which \leq holds;

$xn^2 + yn + z$ is a complexity **upper bound** for $T(n)$

Chapter 2. Getting Started

Chapter 2. Getting Started

Important complexity issues:

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is inaccurate for n to represent the **number of items** in the input.

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is inaccurate for n to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of 2^N , for some very large N .

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is inaccurate for n to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of 2^N , for some very large N .

- If n is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is inaccurate for n to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of 2^N , for some very large N .

- If n is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.
- However, since x_1, x_2, x_3, x_4 are of very large values, a single comparison $x_1 \leq x_2$? would need a **time proportional to N** .

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is inaccurate for n to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of 2^N , for some very large N .

- If n is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.
- However, since x_1, x_2, x_3, x_4 are of very large values, a single comparison $x_1 \leq x_2$? would need a **time proportional to N** .

Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$.

Chapter 2. Getting Started

Important complexity issues:

1. **size of input** n : the number of bits encoding input x , i.e., $n = |x|$.

It is **inaccurate** for n to represent the **number of items** in the input.

Consider to sort 4 items $\langle x_1, x_2, x_3, x_4 \rangle$ of values in the scale of 2^N , for some very large N .

- If n is the number of items, $n = 4$, then any sorting algorithm would run in **constant time**.
- However, since x_1, x_2, x_3, x_4 are of very large values, a single comparison $x_1 \leq x_2$? would need a **time proportional to N** .

Hence, if $n = |\langle x_1, x_2, x_3, x_4 \rangle|$, then $n \approx N$.

To sort the 4 items, a constant number of comparisons is needed, each taking a time linear in N (i.e., **total time is linear in n**).

Chapter 2. Getting Started

Chapter 2. Getting Started

2. Running time $T(n)$: the number of primitive operations executed, a function in n

Chapter 2. Getting Started

2. Running time $T(n)$: the number of primitive operations executed, a function in n

worst-case running time: the running time upper bound for all inputs.

Chapter 2. Getting Started

2. Running time $T(n)$: the number of primitive operations executed, a function in n

worst-case running time: the running time upper bound for all inputs.

order of growth: $T(n) = an^2 + bn + c$ grows the same rate as an^2 (if $a > 0$).

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$

In general,

$$O(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k\}$$

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

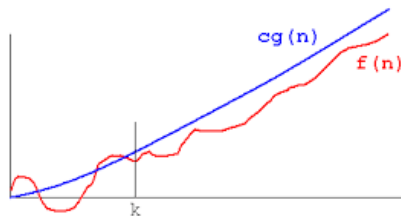
Big-O: set $O(n^2)$ contains all functions of growth rate $\leq cn^2$.

So for function $T(n) = an^2 + bn + c$, $T(n) \in O(n^2)$, but written as

$$T(n) = O(n^2)$$

In general,

$$O(g(n)) = \{f(n) : \exists c > 0, k > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n \geq k\}$$



Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:

Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:

(1) $3n^2$

(2) $5.3n^2 + 6n \log_2 n + 90$

(3) $0.001n^2 - 200n - 5000$

(4) $3n \log_2^n + 6n$

(5) $\sqrt{n} - 20 \log_2 n$

(6) $\log_2 n + 56$

(7) 345

Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:

(1) $3n^2$

(2) $5.3n^2 + 6n \log_2 n + 90$

(3) $0.001n^2 - 200n - 5000$

(4) $3n \log_2^n + 6n$

(5) $\sqrt{n} - 20 \log_2 n$

(6) $\log_2 n + 56$

(7) 345

But the following are not:

Chapter 3. Growth of Functions

For example: the following functions are all of the order of $O(n^2)$:

(1) $3n^2$

(2) $5.3n^2 + 6n \log_2 n + 90$

(3) $0.001n^2 - 200n - 5000$

(4) $3n \log_2^n + 6n$

(5) $\sqrt{n} - 20 \log_2 n$

(6) $\log_2 n + 56$

(7) 345

But the following are not:

(8) $3n^2 \log_2 n - 400n$

(9) $n^{2.001}$

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time** complexity.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

But these are not entirely correct!

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

But these are not entirely correct!

Recall that n should be the **input size**, not number of items, or value

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

But these are not entirely correct!

Recall that n should be the **input size**, not number of items, or value

Assume for INSERTION SORT, the input $\langle x_1, x_2, \dots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \dots, x_m \rangle|$$

INSERTION SORT has time complexity $O(m^2)$

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

But these are not entirely correct!

Recall that n should be the **input size**, not number of items, or value

Assume for INSERTION SORT, the input $\langle x_1, x_2, \dots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \dots, x_m \rangle|$$

INSERTION SORT has time complexity $O(m^2)$

- if every x_i constant B bits, then $m \leq n/B$, the time is $O(n^2)$.

Chapter 3. Growth of Functions

The **Big-O** notation is used to denote **upper bound running time complexity**.

- Algorithm INSERTION SORT has time complexity $O(n^2)$.
- Algorithm ITERATIVE FIBONACCI has time complexity $O(n)$.
- Algorithm REC-FIBONACCI has time complexity $O(2^n)$

But these are not entirely correct!

Recall that n should be the **input size**, not number of items, or value

Assume for INSERTION SORT, the input $\langle x_1, x_2, \dots, x_m \rangle$ has size

$$n = |\langle x_1, x_2, \dots, x_m \rangle|$$

INSERTION SORT has time complexity $O(m^2)$

- if every x_i constant B bits, then $m \leq n/B$, the time is $O(n^2)$.
- if B is not constant, for example, $B = \lceil \log_2 n \rceil$, the time is $O(n^2 / (\log_2^n)^2)$.

Chapter 3. Growth of Functions

Chapter 3. Growth of Functions

Could you also correct the time complexities for INSERTION SORT and REC-FIBONACCI measured by the input size?

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS($A.p, q, r$)

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then
- $t_{1,2} = c$

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then
- $t_{1,2} = c$
- $t_3 = t_4 = T(\frac{n}{2})$

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then
- $t_{1,2} = c$
- $t_3 = t_4 = T(\frac{n}{2})$
- $t_5 \leq n$ (why?)

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q + 1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then
- $t_{1,2} = c$
- $t_3 = t_4 = T(\frac{n}{2})$
- $t_5 \leq n$ (why?)

$$T(n) = t_{1,2} + t_3 + t_4 + t_5 \leq 2T(\frac{n}{2}) + n + c$$

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

A lot of algorithms involve recursions, deriving time complexity has unavoidably resulted in recurrences.

Algorithm MERGE SORT(A, p, r)

1. **if** $p < r$
2. **then** $q = \lfloor \frac{p+r}{2} \rfloor$
3. MERGE SORT(A, p, q)
4. MERGE SORT($A, q+1, r$)
5. MERGING2LISTS(A, p, q, r)

Analysis of the algorithm.

- Assume $n = r - p + 1$, a power of 2;
 also assume $T(n)$ is time for MERGE SORT(A, p, r). Then
- $t_{1,2} = c$
- $t_3 = t_4 = T(\frac{n}{2})$
- $t_5 \leq n$ (why?)

$$T(n) = t_{1,2} + t_3 + t_4 + t_5 \leq 2T(\frac{n}{2}) + n + c$$

base case: $T(1) \leq c$.

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + n + c \\ T\left(\frac{n}{2}\right) &\leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c \end{aligned}$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2, 2², ... to the second, third, ... inequalities, respectively,

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2, 2², ... to the second, third, ... inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying $2, 2^2, \dots$ to the second, third, ... inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2 , 2^2 , \dots to the second, third, \dots inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

$$2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \text{ with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2, 2^2 , ... to the second, third, ... inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

$$2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c$$

...

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2, 2², ... to the second, third, ... inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

$$2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c$$

...

$$2^hT\left(\frac{n}{2^h}\right) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying 2 , 2^2 , \dots to the second, third, \dots inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

$$2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c$$

...

$$2^hT\left(\frac{n}{2^h}\right) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1$$

Chapter 4. Solving Recurrences

Solve recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c \quad \text{with base case } T(1) \leq c$$

with a simple method:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} + c$$

$$T\left(\frac{n}{2^2}\right) \leq 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + c$$

...

$$T\left(\frac{n}{2^h}\right) \leq 2T\left(\frac{n}{2^{h+1}}\right) + \frac{n}{2^h} + c \quad \text{where } \frac{n}{2^{h+1}} = 1$$

multiplying $2, 2^2, \dots$ to the second, third, ... inequalities, respectively,

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n + c$$

$$2T\left(\frac{n}{2}\right) \leq 2^2T\left(\frac{n}{2^2}\right) + 2 \times \frac{n}{2} + 2c$$

$$2^2T\left(\frac{n}{2^2}\right) \leq 2^3T\left(\frac{n}{2^3}\right) + 2^2 \times \frac{n}{2^2} + 2^2c$$

...

$$2^hT\left(\frac{n}{2^h}\right) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + 2^h \times \frac{n}{2^h} + 2^hc \quad \text{where } \frac{n}{2^{h+1}} = 1$$

+

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

Chapter 4. Solving Recurrences

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$T(n) \leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1)$$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \end{aligned}$$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \end{aligned}$$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \\ &= O(n \log_2 n) \end{aligned}$$

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \\ &= O(n \log_2 n) \end{aligned}$$

We need to prove the last equality, i.e., find constants a and k such that

$$n \log_2 n + 2cn - c \leq an \log_2 n \quad (1)$$

when $n > k$.

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \\ &= O(n \log_2 n) \end{aligned}$$

We need to prove the last equality, i.e., find constants a and k such that

$$n \log_2 n + 2cn - c \leq an \log_2 n \quad (1)$$

when $n > k$.

Choose $a = 2$. Then to make (1) holds, we need $\log_2 n > 2c$. So $k = 2^{2c}$ suffices.

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \\ &= O(n \log_2 n) \end{aligned}$$

We need to prove the last equality, i.e., find constants a and k such that

$$n \log_2 n + 2cn - c \leq an \log_2 n \quad (1)$$

when $n > k$.

Choose $a = 2$. Then to make (1) holds, we need $\log_2 n > 2c$. So $k = 2^{2c}$ suffices.

That is, $n \log_2 n + 2cn - c \leq 2n \log_2 n$ when $n > k = 2^{2c}$.

Chapter 4. Solving Recurrences

$$T(n) \leq 2^{h+1}T\left(\frac{n}{2^{h+1}}\right) + (h+1)n + c \sum_{i=0}^h 2^i$$

With $\frac{n}{2^{h+1}} = 1$, we have $n = 2^{h+1}$ or $h+1 = \log_2 n$

$$\begin{aligned} T(n) &\leq 2^{h+1}T(1) + n \log_2 n + c(2^{h+1} - 1) \\ &\leq cn + n \log_2 n + c(n - 1) \\ &= n \log_2 n + 2cn - c \\ &= O(n \log_2 n) \end{aligned}$$

We need to prove the last equality, i.e., find constants a and k such that

$$n \log_2 n + 2cn - c \leq an \log_2 n \quad (1)$$

when $n > k$.

Choose $a = 2$. Then to make (1) holds, we need $\log_2 n > 2c$. So $k = 2^{2c}$ suffices.

That is, $n \log_2 n + 2cn - c \leq 2n \log_2 n$ when $n > k = 2^{2c}$.

So

$$T(n) = O(n \log_2 n)$$

.

Chapter 4. Solving Recurrences

Methods for solving recurrences

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $\mathcal{P}(n)$ for every natural number $n \geq 1$, it suffices to prove

- $\mathcal{P}(1)$ holds;

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $\mathcal{P}(n)$ for every natural number $n \geq 1$, it suffices to prove

- $\mathcal{P}(1)$ holds;
- for every $k \geq 1$, if $\mathcal{P}(k)$ holds, then $\mathcal{P}(k+1)$ holds.

Chapter 4. Solving Recurrences

Methods for solving recurrences

1. Substitution method (based on math induction)

First we recall the principle of the math induction:

To prove a property $\mathcal{P}(n)$ for every natural number $n \geq 1$, it suffices to prove

- $\mathcal{P}(1)$ holds;
- for every $k \geq 1$, if $\mathcal{P}(k)$ holds, then $\mathcal{P}(k+1)$ holds.

"The principle for dominos to fall".



Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k + 1)$ has the variant:

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

- other variants: e.g., $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k)$

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

- other variants: e.g., $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k)$
but we need to make sure all n 's beyond the base cases are covered.

Chapter 4. Solving Recurrences

Math induction comes with different forms or variants

- the base case can be for any integer, e.g., $\mathcal{P}(3)$ instead of $\mathcal{P}(1)$;
- the statement to prove: $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ has the variant:

$$\mathcal{P}(1) \wedge \mathcal{P}(2) \wedge \cdots \wedge \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$$

- other variants: e.g., $\mathcal{P}(k) \longrightarrow \mathcal{P}(2k)$
but we need to make sure all n 's beyond the base cases are covered.

$$\mathcal{P}(k) \longrightarrow \mathcal{P}(2k) \wedge \mathcal{P}(2k+1)$$

Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Proof: (use math induction)

Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Proof: (use math induction)

- Observation: every number should be compared *directly* or *indirectly* with the maximum.

Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Proof: (use math induction)

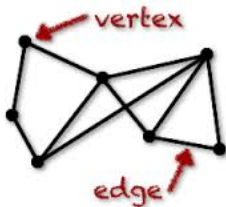
- Observation: every number should be compared *directly* or *indirectly* with the maximum.
- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.

Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Proof: (use math induction)

- Observation: every number should be compared *directly* or *indirectly* with the maximum.
- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.

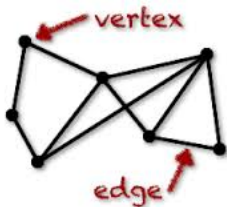


Chapter 4. Solving Recurrences

Theorem. $n - 1$ comparisons are needed to find the maximum of n numbers.

Proof: (use math induction)

- Observation: every number should be compared *directly* or *indirectly* with the maximum.
- the set of numbers can be represented as vertices in a graph, in which edges represent direct comparisons between numbers.



- **Lemma:** every *connected graph* of n vertices has $\geq n - 1$ edges.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1)$$

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1) + r$$

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1) + r = \sum_{i=1}^t m_i - \sum_{i=1}^t 1 + r$$

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?)

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1) + r = \sum_{i=1}^t m_i - \sum_{i=1}^t 1 + r = \sum_{i=1}^t m_i + r - t$$

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?).

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1) + r = \sum_{i=1}^t m_i - \sum_{i=1}^t 1 + r = \sum_{i=1}^t m_i + r - t \geq \sum_{i=1}^t m_i =$$

Chapter 4. Solving Recurrences

Lemma: every *connected graph* of n vertices has $\geq n - 1$ edges.

Proof:

Base case: $n = 1$, a connected graph of single vertex contains zero edges.

Assumption: Assume the lemma holds for every i , $i \leq k$, where $k \geq 1$. That is, a connected graph of i vertices contains at least $i - 1$ edges.

Induction: Let G be any connected graph G of $k + 1$ vertices and v be an arbitrary vertex in it. Assume v shares edges with r other vertices, $r \geq 1$ (why?).

Let G' be the result of removing v from G .

Assume that G' consists of t connected components G_1, \dots, G_t . $1 \leq t \leq r$ (why?).

Let vertex counts of G_1, \dots, G_t be m_1, \dots, m_t , respectively.

Note that $\sum_{i=1}^t m_i = k$ (why?).

By the assumption, these components contain at least $m_1 - 1, \dots, m_t - 1$ edges, respectively.

The total count of edges in G is

$$\geq \sum_{i=1}^t (m_i - 1) + r = \sum_{i=1}^t m_i - \sum_{i=1}^t 1 + r = \sum_{i=1}^t m_i + r - t \geq \sum_{i=1}^t m_i = k$$

Chapter 4. Solving Recurrences

Example: binary search algorithm

Algorithm `BINARY SEARCH`(A, key, i, j)

1. **if** $j < i$ **return** (NULL)
2. **else**
3. $k = \lfloor \frac{i+j}{2} \rfloor$
4. **if** $A[k] = key$ **return** (k)
5. **else**
6. **if** $A[k] > key$ `BINARY SEARCH` ($A, key, i, k - 1$)
7. **else**
8. `BINARY SEARCH` ($A, key, k + 1, j$)

Chapter 4. Solving Recurrences

Example: binary search algorithm

Algorithm `BINARY SEARCH`(A, key, i, j)

1. **if** $j < i$ **return** (NULL)
2. **else**
3. $k = \lfloor \frac{i+j}{2} \rfloor$
4. **if** $A[k] = key$ **return** (k)
5. **else**
6. **if** $A[k] > key$ `BINARY SEARCH` ($A, key, i, k - 1$)
7. **else**
8. `BINARY SEARCH` ($A, key, k + 1, j$)

Let $n = j - i + 1$. It has the time with recurrence,

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

where $c > 0, d > 0$ are constants.

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$
then $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$
then $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$
then $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

$$\text{then } T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$$

Choosing $a \geq c$ allows (2) holds for n (under the assumption (2) holds for $\lfloor \frac{n}{2} \rfloor$)

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \quad \text{when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

$$\text{then } T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$$

Choosing $a \geq c$ allows (2) holds for n (under the assumption (2) holds for $\lfloor \frac{n}{2} \rfloor$)

- Then we need to find the k such that (2) holds for all $n \geq k$.

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

$$\text{then } T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$$

Choosing $a \geq c$ allows (2) holds for n (under the assumption (2) holds for $\lfloor \frac{n}{2} \rfloor$)

- Then we need to find the k such that (2) holds for all $n \geq k$.

$k = 1$ would not work because base case $T(1) = d > 0$ does not guarantee for (2) to hold, because (2) requires $T(1) \leq a \log_2 1 = 0$.

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

$$\text{then } T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$$

Choosing $a \geq c$ allows (2) holds for n (under the assumption (2) holds for $\lfloor \frac{n}{2} \rfloor$)

- Then we need to find the k such that (2) holds for all $n \geq k$.

$k = 1$ would not work because base case $T(1) = d > 0$ does not guarantee for (2) to hold, because (2) requires $T(1) \leq a \log_2 1 = 0$.

$k = 2$ will work because $T(2) \leq T(1) + c = d + c$ guarantee for (2) to hold as long as $d + c \leq a$, as (2) requires $T(2) \leq a \log_2 2$.

Chapter 4. Solving Recurrences

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \text{ and } T(1) = d$$

Prove that $T(n) = O(\log_2 n)$.

Proof. We show that there are constants a, k such that

$$T(n) \leq a \log_2 n \text{ when } n \geq k \quad (2)$$

- First we show that if (2) holds for $\lfloor \frac{n}{2} \rfloor$, i.e., $T(\lfloor \frac{n}{2} \rfloor) \leq a \log_2 \frac{n}{2}$

$$\text{then } T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n + c - a$$

Choosing $a \geq c$ allows (2) holds for n (under the assumption (2) holds for $\lfloor \frac{n}{2} \rfloor$)

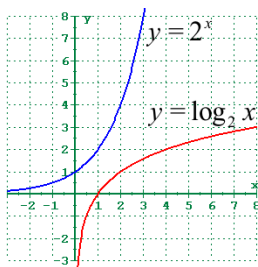
- Then we need to find the k such that (2) holds for all $n \geq k$.

$k = 1$ would not work because base case $T(1) = d > 0$ does not guarantee for (2) to hold, because (2) requires $T(1) \leq a \log_2 1 = 0$.

$k = 2$ will work because $T(2) \leq T(1) + c = d + c$ guarantee for (2) to hold as long as $d + c \leq a$, as (2) requires $T(2) \leq a \log_2 2$.

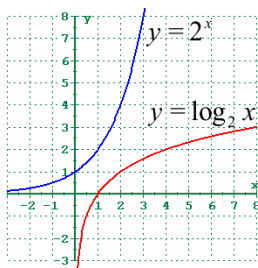
Chapter 4. Solving Recurrences

A little review on logarithm functions:



Chapter 4. Solving Recurrences

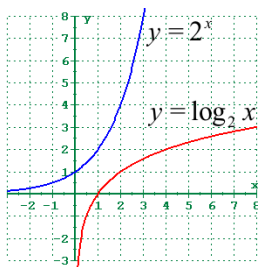
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;

Chapter 4. Solving Recurrences

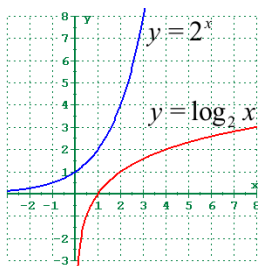
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;

Chapter 4. Solving Recurrences

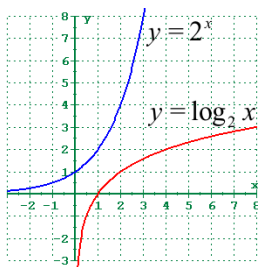
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;

Chapter 4. Solving Recurrences

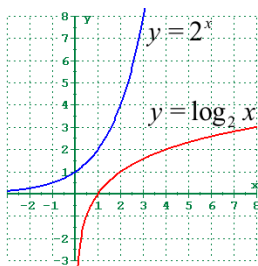
A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;
- $\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n$;

Chapter 4. Solving Recurrences

A little review on logarithm functions:



- $\log_a n + \log_a m = \log_a nm$;
- $\log_a n^b = b \log_a n$, especially $\log_a \frac{1}{n} = -\log_a n$;
- $a^{\log_a n} = n$;
- $\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n$;
- $\log_a^m n = (\log_a n)^m \neq \log_a n^m$.

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

- (1) choose a different base case,

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

- (1) choose a different base case,
- (2) add a constant to the guessed upper bound

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

substitute it in the recurrence, we get

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

substitute it in the recurrence, we get

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

substitute it in the recurrence, we get

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

substitute it in the recurrence, we get

$$\begin{aligned} T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\ &= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n \end{aligned}$$

Chapter 4. Solving Recurrences

Solving recurrence with the substitution method (guess then verify)

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n, \text{ where } T(1) = 2$$

Guess $T(n) \leq cn \log_2 n$, for some constant c to be determined later.

Verify:

- base case: $T(1) = 2 \leq 0$ **does not** hold for the guessed inequality.

There are two remedies:

(1) choose a different base case, (2) add a constant to the guessed upper bound

We instead **guess** $T(n) \leq cn \log_2 n + 2$, so it holds for $n = 1$.

Now **assume the guessed upper bound holds for** $\lfloor \frac{2n}{3} \rfloor$, i.e.,

$$T(\lfloor \frac{2n}{3} \rfloor) \leq c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2$$

substitute it in the recurrence, we get

$$T(n) = \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c \lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n$$

$$= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2}$$

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound,

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound, we have choose c such that $1 + n - cn \log_2 \frac{3}{2} \leq 0$.

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound, we have choose c such that $1 + n - cn \log_2 \frac{3}{2} \leq 0$.

for example, we can choose $c = 2/\log_2 \frac{3}{2}$, such that

$$1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0$$

for all $n \geq 1$.

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + 1 + n - cn \log_2 \frac{3}{2}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound, we have choose c such that $1 + n - cn \log_2 \frac{3}{2} \leq 0$.

for example, we can choose $c = 2/\log_2 \frac{3}{2}$, such that

$$1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0$$

for all $n \geq 1$.

So we have shown $T(n) \leq cn \log_2 n + 2$, for all $n \geq 1$.

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + \textcolor{brown}{1} + \textcolor{brown}{n} - \textcolor{brown}{cn} \log_2 \frac{3}{2}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound, we have choose c such that $\textcolor{brown}{1} + \textcolor{brown}{n} - \textcolor{brown}{cn} \log_2 \frac{3}{2} \leq 0$.

for example, we can choose $c = 2/\log_2 \frac{3}{2}$, such that

$$1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0$$

for all $n \geq 1$.

So we have shown $T(n) \leq cn \log_2 n + 2$, for all $n \geq 1$. [Are you sure?](#)

Chapter 4. Solving Recurrences

$$\begin{aligned}T(n) &= \frac{3}{2}T(\lfloor \frac{2n}{3} \rfloor) + n \leq \frac{3}{2}(c\lfloor \frac{2n}{3} \rfloor \log_2 \lfloor \frac{2n}{3} \rfloor + 2) + n \leq cn \log_2 \frac{2n}{3} + 3 + n \\&= cn(\log_2 n + \log_2 \frac{2}{3}) + 3 + n = cn \log_2 n + 3 + n - cn \log_2 \frac{3}{2} \\&= \leq cn \log_2 n + 2 + \mathbf{1 + n - cn \log_2 \frac{3}{2}}\end{aligned}$$

To allow that the last term $\leq cn \log_2 n + 2$, the assumed upper bound, we have choose c such that $\mathbf{1 + n - cn \log_2 \frac{3}{2}} \leq 0$.

for example, we can choose $c = 2/\log_2 \frac{3}{2}$, such that

$$1 + n - cn \log_2 \frac{3}{2} = 1 + n - 2n = 1 - n \leq 0$$

for all $n \geq 1$.

So we have shown $T(n) \leq cn \log_2 n + 2$, for all $n \geq 1$. **Are you sure?**

Does this also imply

$$T(n) = O(n \log_2 n) \text{ or } T(n) = O(n \log_2 n + 2) ?$$

Chapter 4. Solving Recurrences

2. Changing variables

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$,

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m$$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m$$

rename the function: $S(m) = T(2^m)$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m$$

rename the function: $S(m) = T(2^m)$

$$S(m) = 2S(m/2) + m$$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m$$

rename the function: $S(m) = T(2^m)$

$$S(m) = 2S(m/2) + m$$

solve it, we have $S(m) = O(m \log m)$

Chapter 4. Solving Recurrences

2. Changing variables

Example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

Define $m = \log_2 n$, i.e., $n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m$$

rename the function: $S(m) = T(2^m)$

$$S(m) = 2S(m/2) + m$$

solve it, we have $S(m) = O(m \log m)$

so $T(n) = T(2^m) = O(m \log m) = O(\log n \log \log n)$.

Chapter 4. Solving Recurrences

3. Recursive tree method

Chapter 4. Solving Recurrences

3. Recursive tree method

By **unfolding** the recurrence to make a recursive-tree.

Chapter 4. Solving Recurrences

3. Recursive tree method

By **unfolding** the recurrence to make a recursive-tree.

(1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.

Chapter 4. Solving Recurrences

3. Recursive tree method

By **unfolding** the recurrence to make a recursive-tree.

- (1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.
- (2) for each child, replace it with then non-recursive terms and produce children that are then recursive terms

Chapter 4. Solving Recurrences

3. Recursive tree method

By **unfolding** the recurrence to make a recursive-tree.

- (1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.
- (2) for each child, replace it with then non-recursive terms and produce children that are then recursive terms
- (3) repeat (2), expand the tree until all children are the base case.

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

l_0 :

$T(n)$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0:$		$T(n)$		n^2
$l_1:$	$T(n/4)$	$T(n/4)$	$T(n/4)$	

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllllll} l_0: & & & & T(n) & & & n^2 \\ l_1: & & T(n/4) & & T(n/4) & & T(n/4) & 3(\frac{n}{4})^2 \\ l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & \end{array}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll} l_0: & & & T(n) & & n^2 \\ l_1: & T(n/4) & & T(n/4) & & 3\left(\frac{n}{4}\right)^2 \\ l_2: & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & 3^2\left(\frac{n}{4^2}\right)^2 \\ l_3: & \dots\dots\dots \end{array}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$l_0:$				$T(n)$				n^2
$l_1:$		$T(n/4)$		$T(n/4)$		$T(n/4)$		$3(\frac{n}{4})^2$
$l_2:$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$3^2(\frac{n}{4^2})^2$
$l_3:$							$3^3(\frac{n}{4^3})^2$
$l_4:$							

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

l_0 :				$T(n)$				n^2
l_1 :		$T(n/4)$		$T(n/4)$		$T(n/4)$		$3(\frac{n}{4})^2$
l_2 :	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$T(\frac{n}{4^2})$	$3^2(\frac{n}{4^2})^2$
l_3 :							$3^3(\frac{n}{4^3})^2$
l_4 :							
l_{m-1} :							

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll} l_0: & & & & T(n) & n^2 \\ l_1: & T(n/4) & & T(n/4) & T(n/4) & 3\left(\frac{n}{4}\right)^2 \\ l_2: & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & 3^2\left(\frac{n}{4^2}\right)^2 \\ l_3: & \dots\dots\dots & & & & & & & 3^3\left(\frac{n}{4^3}\right)^2 \\ l_4: & \dots\dots\dots & & & & & & & \\ l_{m-1}: & \dots\dots\dots & & & & & & & 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \end{array}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & T(n) & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3\left(\frac{n}{4}\right)^2 \\
 l_2: & T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) T\left(\frac{n}{4^2}\right) & & 3^2\left(\frac{n}{4^2}\right)^2 \\
 l_3: & \dots\dots\dots & & & & 3^3\left(\frac{n}{4^3}\right)^2 \\
 l_4: & \dots\dots\dots & & & & \\
 l_{m-1}: & \dots\dots\dots & & & & 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & T(1)
 \end{array}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & T(n) & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3\left(\frac{n}{4}\right)^2 \\
 l_2: & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & 3^2\left(\frac{n}{4^2}\right)^2 \\
 l_3: & \dots\dots & & & & & 3^3\left(\frac{n}{4^3}\right)^2 \\
 l_4: & \dots\dots & & & & & \\
 l_{m-1}: & \dots\dots & & & & & 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & T(n) & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3\left(\frac{n}{4}\right)^2 \\
 l_2: & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & 3^2\left(\frac{n}{4^2}\right)^2 \\
 l_3: & \dots\dots\dots & & & & & 3^3\left(\frac{n}{4^3}\right)^2 \\
 l_4: & \dots\dots\dots & & & & & \\
 l_{m-1}: & \dots\dots\dots & & & & & 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \\
 l_m: & \color{blue}{T(1), T(1), T(1), T(1), \dots,} & & & & & \color{blue}{T(1)}
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$T(n) = n^2 \left[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \dots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2 \right] + \color{blue}{3^m T(1)}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & & T(n) & n^2 \\
 l_1: & & T(n/4) & & T(n/4) & & 3\left(\frac{n}{4}\right)^2 \\
 l_2: & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & T\left(\frac{n}{4^2}\right) & 3^2\left(\frac{n}{4^2}\right)^2 \\
 l_3: & \dots\dots & & & & & 3^3\left(\frac{n}{4^3}\right)^2 \\
 l_4: & \dots\dots & & & & & \\
 l_{m-1}: & \dots\dots & & & & & 3^{m-1}\left(\frac{n}{4^{m-1}}\right)^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2\left[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \dots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2\right] + 3^m T(1) \\
 T(n) &= n^2\left[1 + 3\left(\frac{1}{4}\right)^2 + 3^2\left(\frac{1}{4^2}\right)^2 + 3^3\left(\frac{1}{4^3}\right)^2 + \dots + 3^{m-1}\left(\frac{1}{4^{m-1}}\right)^2\right] + 3^m \times 1
 \end{aligned}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & T(n) & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3(\frac{n}{4})^2 \\
 l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & 3^2(\frac{n}{4^2})^2 \\
 l_3: & \dots\dots & & & & & 3^3(\frac{n}{4^3})^2 \\
 l_4: & \dots\dots & & & & & \\
 l_{m-1}: & \dots\dots & & & & & 3^{m-1}(\frac{n}{4^{m-1}})^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
 &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m (\frac{n}{4^m})^2
 \end{aligned}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & & T(n) & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3(\frac{n}{4})^2 \\
 l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & 3^2(\frac{n}{4^2})^2 \\
 l_3: & \dots\dots & & & & & 3^3(\frac{n}{4^3})^2 \\
 l_4: & \dots\dots & & & & & \\
 l_{m-1}: & \dots\dots & & & & & 3^{m-1}(\frac{n}{4^{m-1}})^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
 &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{n}{4^{m-1}})^2] + 3^m (\frac{n}{4^m})^2 \\
 &= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \dots + (\frac{3}{16})^m]
 \end{aligned}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llllll}
 l_0: & & & T(n) & & n^2 \\
 l_1: & T(n/4) & & T(n/4) & T(n/4) & 3(\frac{n}{4})^2 \\
 l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & 3^2(\frac{n}{4^2})^2 \\
 l_3: & \dots\dots & & & & & 3^3(\frac{n}{4^3})^2 \\
 l_4: & \dots\dots & & & & & \\
 l_{m-1}: & \dots\dots & & & & & 3^{m-1}(\frac{n}{4^{m-1}})^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
 &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{n}{4^{m-1}})^2] + 3^m (\frac{n}{4^m})^2 \\
 &= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \dots + (\frac{3}{16})^m] \\
 &= n^2(\frac{1 - (\frac{3}{16})^{m+1}}{1 - \frac{3}{16}})
 \end{aligned}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llll}
 l_0: & & T(n) & n^2 \\
 l_1: & T(n/4) & T(n/4) & 3(\frac{n}{4})^2 \\
 l_2: & T(\frac{n}{4^2}) \ T(\frac{n}{4^2}) \ T(\frac{n}{4^2}) & T(\frac{n}{4^2}) \ T(\frac{n}{4^2}) \ T(\frac{n}{4^2}) & 3^2(\frac{n}{4^2})^2 \\
 l_3: & \dots\dots & & 3^3(\frac{n}{4^3})^2 \\
 l_4: & \dots\dots & & \\
 l_{m-1}: & \dots\dots & & 3^{m-1}(\frac{n}{4^{m-1}})^2 \\
 l_m: & \textcolor{blue}{T(1), T(1), T(1), T(1), \dots,} & & \textcolor{blue}{T(1)}
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + \textcolor{blue}{3^m T(1)} \\
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + \textcolor{blue}{3^m \times 1} \\
 &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{n}{4^{m-1}})^2] + \textcolor{blue}{3^m (\frac{n}{4^m})^2} \\
 &= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \dots + (\frac{3}{16})^m] \\
 &= n^2(\frac{1 - (\frac{3}{16})^{m+1}}{1 - \frac{3}{16}}) \\
 &\leq n^2(\frac{1}{1 - \frac{3}{16}})
 \end{aligned}$$

Chapter 4. Solving Recurrences

Example $T(n) = 3T(n/4) + n^2$, with base case $T(1) = 1$

$$\begin{array}{llll}
 l_0: & & T(n) & n^2 \\
 l_1: & T(n/4) & T(n/4) & 3(\frac{n}{4})^2 \\
 l_2: & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & T(\frac{n}{4^2}) & 3^2(\frac{n}{4^2})^2 \\
 l_3: & \dots\dots & & & 3^3(\frac{n}{4^3})^2 \\
 l_4: & \dots\dots & & & \\
 l_{m-1}: & \dots\dots & & & 3^{m-1}(\frac{n}{4^{m-1}})^2 \\
 l_m: & T(1), T(1), T(1), T(1), \dots, & & & T(1)
 \end{array}$$

where $\frac{n}{4^m} = 1$, i.e., $m = \log_4 n$.

Then $T(n)$ is the sum

$$\begin{aligned}
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m T(1) \\
 T(n) &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{1}{4^{m-1}})^2] + 3^m \times 1 \\
 &= n^2[1 + 3(\frac{1}{4})^2 + 3^2(\frac{1}{4^2})^2 + 3^3(\frac{1}{4^3})^2 + \dots + 3^{m-1}(\frac{n}{4^{m-1}})^2] + 3^m (\frac{n}{4^m})^2 \\
 &= n^2[1 + \frac{3}{16} + (\frac{3}{16})^2 + (\frac{3}{16})^3 + \dots + (\frac{3}{16})^m] \\
 &= n^2(\frac{1 - (\frac{3}{16})^{m+1}}{1 - \frac{3}{16}}) \\
 &\leq n^2(\frac{1}{1 - \frac{3}{16}}) \\
 &= \frac{16}{13} n^2
 \end{aligned}$$

for all $n > 0$.

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

l_0 :

$$n^2$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$l_0:$		n^2	
$l_1:$	$(n/4)^2$	$(n/4)^2$	$(n/4)^2$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

l_0 :

$$n^2$$

l_1 :

$$(n/4)^2$$

$$(n/4)^2$$

$$(n/4)^2$$

l_2 :

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

$$(\frac{n}{4^2})^2$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{l} l_0: \qquad \qquad \qquad n^2 \\ l_1: \qquad \qquad (n/4)^2 \qquad \qquad (n/4)^2 \qquad \qquad (n/4)^2 \\ l_2: \quad \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \quad \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \quad \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \left(\frac{n}{4^2}\right)^2 \\ l_3: \dots\dots \end{array}$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{llll} l_0: & & n^2 & \\ l_1: & (n/4)^2 & (n/4)^2 & (n/4)^2 \\ l_2: & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 \\ l_3: & \dots\dots\dots & & 3^3 \text{ nodes of } (\frac{n}{4^3})^2 \\ l_4: & \dots\dots\dots & & \end{array}$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{llll} l_0: & & n^2 & \\ l_1: & (n/4)^2 & (n/4)^2 & (n/4)^2 \\ l_2: & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 \\ l_3: & \dots\dots & & 3^3 \text{ nodes of } (\frac{n}{4^3})^2 \\ l_4: & \dots\dots & & \\ l_{m-1}: & \dots\dots & & \end{array}$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{llll} l_0: & & n^2 & \\ l_1: & (n/4)^2 & (n/4)^2 & (n/4)^2 \\ l_2: & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 \\ l_3: & \dots\dots & & 3^3 \text{ nodes of } (\frac{n}{4^3})^2 \\ l_4: & \dots\dots & & \\ l_{m-1}: & \dots\dots & & 3^{m-1} \text{ of } (\frac{n}{4^{m-1}})^2 \end{array}$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{llll} l_0: & & n^2 & \\ l_1: & (n/4)^2 & (n/4)^2 & (n/4)^2 \\ l_2: & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 \\ l_3: & \dots\dots\dots & & 3^3 \text{ nodes of } (\frac{n}{4^3})^2 \\ l_4: & \dots\dots\dots & & \\ l_{m-1}: & \dots\dots\dots & & 3^{m-1} \text{ of } (\frac{n}{4^{m-1}})^2 \\ l_m: & T(1), T(1), T(1), T(1), T(1), \dots, & & 3^m \text{ nodes of } T(1) \end{array}$$

Chapter 4. Solving Recurrences

Recursive tree notation in the textbook:

$$T(n) = 3T(n/4) + n^2, \text{ with base case } T(1) = 1$$

$$\begin{array}{llll} l_0: & & n^2 & \\ l_1: & (n/4)^2 & (n/4)^2 & (n/4)^2 \\ l_2: & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 & (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 (\frac{n}{4^2})^2 \\ l_3: & \dots\dots\dots & & 3^3 \text{ nodes of } (\frac{n}{4^3})^2 \\ l_4: & \dots\dots\dots & & \\ l_{m-1}: & \dots\dots\dots & & 3^{m-1} \text{ of } (\frac{n}{4^{m-1}})^2 \\ l_m: & T(1), T(1), T(1), T(1), T(1), \dots, & & 3^m \text{ nodes of } T(1) \end{array}$$

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;
- With assumption that input data are in a probabilistic distribution

Chapter 5. Probabilistic Analysis of Algorithms

Chapter 5. Probabilistic analysis and randomized algorithms

- Estimate efficiency of algorithms on a majority of inputs, not all inputs;
- Performance is “average” cases, not the worst case;
- With assumption that input data are in a probabilistic distribution
- Close relationship with randomized algorithms

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.
in other word, it is efficient in most cases;

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

in other word, it is efficient in most cases;

assumption: the input data are of the uniform distribution.

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

in other word, it is efficient in most cases;

assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

in other word, it is efficient in most cases;

assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.

Alternatively, we can enforce the desired distribution by using randomness (tossing coins) in the algorithms.

Chapter 5. Probabilistic Analysis of Algorithms

A good example: QUICK SORT algorithm [Hoare'1959]

- It has the worst case time $T_{wc}(n) \geq an^2$ for some constant $a > 0$.
- It has the average case time $T_{ac}(n) \leq bn \log_2 n$ for a constant $b > 0$.

in other word, it is efficient in most cases;

assumption: the input data are of the uniform distribution.

- But usually the input data do not satisfy uniform distribution.

Alternatively, we can enforce the desired distribution by using randomness (tossing coins) in the algorithms.

that is, we use randomized algorithms.

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**
- such random events are driven by random data

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**
- such random events are driven by random data
which are either input data or randomly tossed coins

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**
- such random events are driven by random data
which are either input data or randomly tossed coins
- So the running time come with a probability

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**
- such random events are driven by random data
which are either input data or randomly tossed coins
- So the running time come with a probability
you can compute the expected time (i.e., averaged time)

Chapter 5. Probabilistic Analysis of Algorithms

In both probabilistic analysis of deterministic algorithms
and analysis of randomized algorithms

- actions in the algorithm are considered **random events**
- such random events are driven by random data
which are either input data or randomly tossed coins
- So the running time come with a probability
you can compute the expected time (i.e., averaged time)

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $Prob(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $Prob(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;
 $Prob(\text{to answer 'NO' on 'NO' instance}) = 1$

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $\text{Prob}(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;
 $\text{Prob}(\text{to answer 'NO' on 'NO' instance}) = 1$
 - on 'YES' instances, $\geq 75\%$ accuracy;

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $Prob(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;
 $Prob(\text{to answer 'NO' on 'NO' instance}) = 1$
 - on 'YES' instances, $\geq 75\%$ accuracy;
 $Prob(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $Prob(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;
 $Prob(\text{to answer 'NO' on 'NO' instance}) = 1$
 - on 'YES' instances, $\geq 75\%$ accuracy;
 $Prob(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$
Accuracy 75% can be improved to 99.99% with multiple trials.

Chapter 5. Probabilistic Analysis of Algorithms

There are two types of randomized algorithms:

- Las Vegas algorithms
 - always gives answer correctly;
 - running time comes with a probability distribution
e.g., for QUICKSORT, $Prob(T(n) \leq cn \log n) \geq 0.75$
- Monte Carlo algorithms
 - on 'NO' instances, 100% accuracy;
 $Prob(\text{to answer 'NO' on 'NO' instance}) = 1$
 - on 'YES' instances, $\geq 75\%$ accuracy;
 $Prob(\text{to answer 'YES' on 'YES' instance}) \geq 0.75$
Accuracy 75% can be improved to 99.99% with multiple trials.
- Las Vegas algorithms is as powerful as Monte Carlo algorithms, if not more.