Chapter 15

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.

```
ARG seed 1
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
  Base   : 0x0000363c (decimal 13884)
  Limit  : 290
Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  1: 0x00000105 (decimal:  261) --> VALID: 0x00003741 (decimal: 13884+261 = 14145)
  VA  2: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  3: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  4: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION

ARG seed 2
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500
Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> VALID: 0x00003ce2 (decimal: 15529+57 = 15586)
  VA  1: 0x00000056 (decimal:   86) --> VALID: 0x00003cff (decimal: 15529+86 = 15615)
  VA  2: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION

ARG seed 3
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316
Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> SEGMENTATION VIOLATION
  VA  1: 0x0000026a (decimal:  618) --> SEGMENTATION VIOLATION
  VA  2: 0x00000280 (decimal:  640) --> SEGMENTATION VIOLATION
  VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8916 + 67 = 8983)
  VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8916 + 13 = 8929)
```

2. Run with these flags: -s 0 -n 10. What value do you have to set -l (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?

-l should be 930 or greater as the largest VA is 929

relocation.py -s 0 -n 10 -l 930 -c

```
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
  Base   : 0x0000360b (decimal 13835)
  Limit  : 930
```

```
Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> VALID: 0x000039ac (decimal: 14764) largest
```

3. Run with these flags: -s 1 -n 10 -l 100. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?

The base register + limit must fit entirely within physical memory.

so

base + limit - 1 <= max physical address, where max physical address = 16k -1 = 16383

base <= 16383 - 99 = 16284

```
ARG seed 1
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100
Virtual Address Trace
  VA  0: 0x00000089 (decimal:  137) --> SEGMENTATION VIOLATION
  VA  1: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  2: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  3: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  4: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  5: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  6: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  7: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  8: 0x00000060 (decimal:   96) --> VALID: 0x00003ffc (decimal: 16380)
  VA  9: 0x0000001d (decimal:   29) --> VALID: 0x00003fb9 (decimal: 16313)
```

test if base = 16285

```
Base-and-Bounds register information:
  Base   : 0x00003f9d (decimal 16285)
  Limit  : 100
Error: address space does not fit into physical memory with those base/bounds values.
Base + Limit: 16385   Psize: 16384
```

4. Run some of the same problems above, but with larger address spaces (-a) and physical memories (-p).

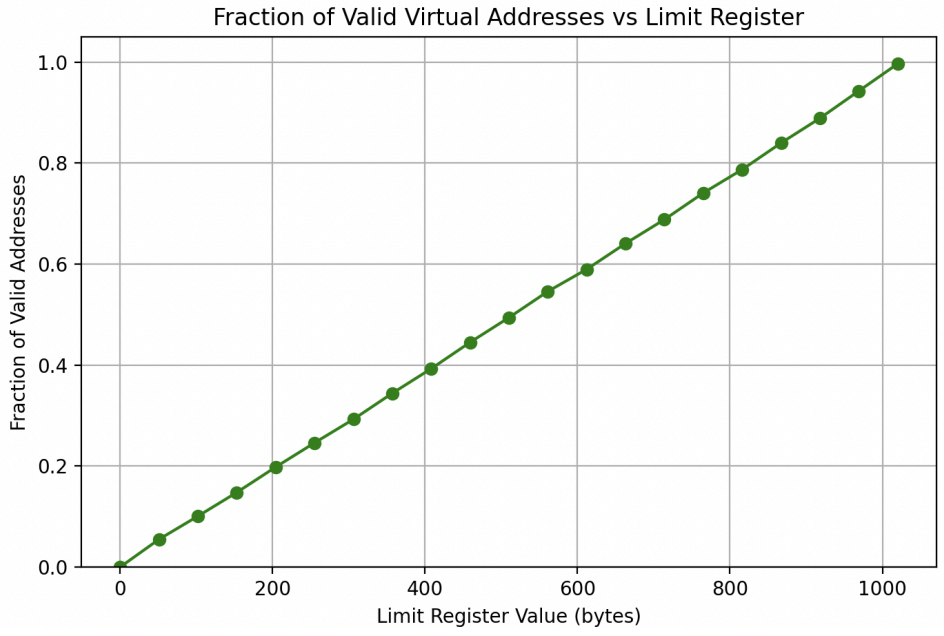
```
relocation.py -s 1 -n 10 -a 2k -p 32k -c
ARG seed 1
ARG address space size 2k
ARG phys mem size 32k
Base-and-Bounds register information:
  Base   : 0x00006c78 (decimal 27768)
  Limit  : 580
Virtual Address Trace
  VA  0: 0x0000061c (decimal: 1564) --> SEGMENTATION VIOLATION
  VA  1: 0x0000020a (decimal:  522) --> VALID: 0x00006e82 (decimal: 28290)
  VA  2: 0x000003f6 (decimal: 1014) --> SEGMENTATION VIOLATION
  VA  3: 0x00000398 (decimal:  920) --> SEGMENTATION VIOLATION
  VA  4: 0x00000536 (decimal: 1334) --> SEGMENTATION VIOLATION
  VA  5: 0x0000064f (decimal: 1615) --> SEGMENTATION VIOLATION
  VA  6: 0x000000c0 (decimal:  192) --> VALID: 0x00006d38 (decimal: 27960)
  VA  7: 0x0000003a (decimal:   58) --> VALID: 0x00006cb2 (decimal: 27826)
  VA  8: 0x000006af (decimal: 1711) --> SEGMENTATION VIOLATION
  VA  9: 0x00000376 (decimal:  886) --> SEGMENTATION VIOLATION
```

I found: 1. segmentation violations occur for any VA ≥ limit, regardless of physical memory size. 2. physical memory size only affects how high the base can be placed.

5. What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

If L = 0 → fraction = 0 (no address is valid)
If L = VA_max → fraction = 1 (all addresses valid)

code relocation_graph.py is github



Chapter 16

1. First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses? segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
For upward-growing segments:
PA = base + offset
For downward-growing segments:
PA = base - 1 - offset & offset = VA_max - VA

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20
  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20
Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  512 - 127 - 108 - 1  = 492)
  VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)


ARG seed 1
ARG address space size 128
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20
  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20
Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   0 + 17 = 17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  512 - 127 - 108 - 1 = 492)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20
  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20
Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  512 – 127 - 122 - 1 = 506)
  VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  512 - 127 - 121 - 1 = 505)
```

```
  VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    0 + 7 = 7)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   0 10 = 10)
  VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

2. Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest illegal addresses in this entire address space? Finally, how would you run segmentation.py with the -A flag to test if you are right?

Segment 0 has limit 20, meaning offsets 0-19 are valid
Lowest legal VA in SEG1 = 127 - 19 = 108 (0x6C)
Lowest illegal address = 20
Highest illegal address = 107 (0x6B)

```
python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A 19,20,107,108 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20
  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20
Virtual Address Trace
  VA  0: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  1: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

3. Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters: segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 ? --l0 ? --b1 ? --l1 ?

Segment 0 Setup:
Need VAs 0 and 1 valid, VAs 2-7 invalid
limit0 = 2 (allows offsets 0-1)
base0 = 0 (can be any valid address in physical memory)

Segment 1 Setup:
Need VAs 14 and 15 valid, VAs 8-13 invalid
For downward-growing: offset = VA_max - VA = 15 - VA
VA 15: offset = 0 (valid)
VA 14: offset = 1 (valid)
VA 13: offset = 2 (should be invalid)
limit1 = 2 (allows offsets 0-1)
base1 = 128 (can be any valid address in physical memory)

```
python3 segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 128 --l1 2 -c
ARG seed 0
```

```
ARG address space size 16
ARG phys mem size 128
Segment register information:
  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 2
  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 2
Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007f (decimal:  127)
```

4. Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?

To achieve 90% valid addresses in an address space of size a (where Segment 0 covers VAs 0 to a/2-1 and Segment 1 covers VAs a/2 to a-1), set the segment limits so that limit0 + limit1 = 0.9 × a.

The critical parameters are:
-l (--l0): Segment 0 limit
-L (--l1): Segment 1 limit
-a: Address space size

For example
Address space = 100, need 90% valid = 90 addresses
python3 segmentation.py -s 3 -a 100 -p 512 -l 45 -L 45 -n 100 -c

with random seed set to 3, we have 8 seg violation over 100 VAs, so roughly 90%

```
python3 segmentation.py -s 3 -a 100 -p 512 -l 45 -L 45 -n 100 -c
ARG seed 3
ARG address space size 100
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x00000079 (decimal 121)
  Segment 0 limit                  : 45
  Segment 1 base  (grows negative) : 0x00000143 (decimal 323)
  Segment 1 limit                  : 45
Virtual Address Trace
  VA  0: 0x00000024 (decimal:   36) --> VALID in SEG0: 0x0000009d (decimal:  157)
  VA  1: 0x0000003c (decimal:   60) --> VALID in SEG1: 0x0000011b (decimal:  283)
```

VA 2: 0x0000003e (decimal: 62) --> VALID in SEG1: 0x0000011d (decimal: 285)
VA 3: 0x00000006 (decimal: 6) --> VALID in SEG0: 0x0000007f (decimal: 127)
VA 4: 0x00000001 (decimal: 1) --> VALID in SEG0: 0x0000007a (decimal: 122)
VA 5: 0x00000053 (decimal: 83) --> VALID in SEG1: 0x00000132 (decimal: 306)
VA 6: 0x00000019 (decimal: 25) --> VALID in SEG0: 0x00000092 (decimal: 146)
VA 7: 0x00000017 (decimal: 23) --> VALID in SEG0: 0x00000090 (decimal: 144)
VA 8: 0x00000063 (decimal: 99) --> VALID in SEG1: 0x00000142 (decimal: 322)
VA 9: 0x0000002f (decimal: 47) --> SEGMENTATION VIOLATION (SEG0)
VA 10: 0x00000053 (decimal: 83) --> VALID in SEG1: 0x00000132 (decimal: 306)
VA 11: 0x0000002f (decimal: 47) --> SEGMENTATION VIOLATION (SEG0)
VA 12: 0x0000003f (decimal: 63) --> VALID in SEG1: 0x0000011e (decimal: 286)
VA 13: 0x0000000f (decimal: 15) --> VALID in SEG0: 0x00000088 (decimal: 136)
VA 14: 0x0000003f (decimal: 63) --> VALID in SEG1: 0x0000011e (decimal: 286)
VA 15: 0x00000056 (decimal: 86) --> VALID in SEG1: 0x00000135 (decimal: 309)
VA 16: 0x00000034 (decimal: 52) --> SEGMENTATION VIOLATION (SEG1)
VA 17: 0x0000004a (decimal: 74) --> VALID in SEG1: 0x00000129 (decimal: 297)
VA 18: 0x00000043 (decimal: 67) --> VALID in SEG1: 0x00000122 (decimal: 290)
VA 19: 0x00000006 (decimal: 6) --> VALID in SEG0: 0x0000007f (decimal: 127)
VA 20: 0x0000004b (decimal: 75) --> VALID in SEG1: 0x0000012a (decimal: 298)
VA 21: 0x0000003b (decimal: 59) --> VALID in SEG1: 0x0000011a (decimal: 282)
VA 22: 0x0000001e (decimal: 30) --> VALID in SEG0: 0x00000097 (decimal: 151)
VA 23: 0x00000003 (decimal: 3) --> VALID in SEG0: 0x0000007c (decimal: 124)
VA 24: 0x00000056 (decimal: 86) --> VALID in SEG1: 0x00000135 (decimal: 309)
VA 25: 0x0000002f (decimal: 47) --> SEGMENTATION VIOLATION (SEG0)
VA 26: 0x00000047 (decimal: 71) --> VALID in SEG1: 0x00000126 (decimal: 294)
VA 27: 0x00000057 (decimal: 87) --> VALID in SEG1: 0x00000136 (decimal: 310)
VA 28: 0x00000047 (decimal: 71) --> VALID in SEG1: 0x00000126 (decimal: 294)
VA 29: 0x0000005c (decimal: 92) --> VALID in SEG1: 0x0000013b (decimal: 315)
VA 30: 0x00000027 (decimal: 39) --> VALID in SEG0: 0x000000a0 (decimal: 160)
VA 31: 0x00000050 (decimal: 80) --> VALID in SEG1: 0x0000012f (decimal: 303)
VA 32: 0x0000002c (decimal: 44) --> VALID in SEG0: 0x000000a5 (decimal: 165)
VA 33: 0x0000005d (decimal: 93) --> VALID in SEG1: 0x0000013c (decimal: 316)
VA 34: 0x00000057 (decimal: 87) --> VALID in SEG1: 0x00000136 (decimal: 310)
VA 35: 0x00000009 (decimal: 9) --> VALID in SEG0: 0x00000082 (decimal: 130)
VA 36: 0x0000000d (decimal: 13) --> VALID in SEG0: 0x00000086 (decimal: 134)
VA 37: 0x00000015 (decimal: 21) --> VALID in SEG0: 0x0000008e (decimal: 142)
VA 38: 0x00000060 (decimal: 96) --> VALID in SEG1: 0x0000013f (decimal: 319)
VA 39: 0x0000002b (decimal: 43) --> VALID in SEG0: 0x000000a4 (decimal: 164)
VA 40: 0x0000003e (decimal: 62) --> VALID in SEG1: 0x0000011d (decimal: 285)
VA 41: 0x0000001e (decimal: 30) --> VALID in SEG0: 0x00000097 (decimal: 151)
VA 42: 0x00000032 (decimal: 50) --> SEGMENTATION VIOLATION (SEG1)
VA 43: 0x00000026 (decimal: 38) --> VALID in SEG0: 0x0000009f (decimal: 159)
VA 44: 0x00000023 (decimal: 35) --> VALID in SEG0: 0x0000009c (decimal: 156)
VA 45: 0x0000003a (decimal: 58) --> VALID in SEG1: 0x00000119 (decimal: 281)
VA 46: 0x0000003a (decimal: 58) --> VALID in SEG1: 0x00000119 (decimal: 281)
VA 47: 0x0000005a (decimal: 90) --> VALID in SEG1: 0x00000139 (decimal: 313)
VA 48: 0x00000044 (decimal: 68) --> VALID in SEG1: 0x00000123 (decimal: 291)
VA 49: 0x0000005c (decimal: 92) --> VALID in SEG1: 0x0000013b (decimal: 315)
VA 50: 0x00000055 (decimal: 85) --> VALID in SEG1: 0x00000134 (decimal: 308)
VA 51: 0x00000063 (decimal: 99) --> VALID in SEG1: 0x00000142 (decimal: 322)
VA 52: 0x00000043 (decimal: 67) --> VALID in SEG1: 0x00000122 (decimal: 290)
VA 53: 0x00000010 (decimal: 16) --> VALID in SEG0: 0x00000089 (decimal: 137)
VA 54: 0x00000056 (decimal: 86) --> VALID in SEG1: 0x00000135 (decimal: 309)
VA 55: 0x00000060 (decimal: 96) --> VALID in SEG1: 0x0000013f (decimal: 319)
VA 56: 0x0000005a (decimal: 90) --> VALID in SEG1: 0x00000139 (decimal: 313)
VA 57: 0x00000038 (decimal: 56) --> VALID in SEG1: 0x00000117 (decimal: 279)
VA 58: 0x00000047 (decimal: 71) --> VALID in SEG1: 0x00000126 (decimal: 294)
VA 59: 0x00000015 (decimal: 21) --> VALID in SEG0: 0x0000008e (decimal: 142)
VA 60: 0x00000053 (decimal: 83) --> VALID in SEG1: 0x00000132 (decimal: 306)
VA 61: 0x00000039 (decimal: 57) --> VALID in SEG1: 0x00000118 (decimal: 280)
VA 62: 0x0000001c (decimal: 28) --> VALID in SEG0: 0x00000095 (decimal: 149)
VA 63: 0x00000006 (decimal: 6) --> VALID in SEG0: 0x0000007f (decimal: 127)

```
  VA 64: 0x00000055 (decimal:   85) --> VALID in SEG1: 0x00000134 (decimal:  308)
  VA 65: 0x00000062 (decimal:   98) --> VALID in SEG1: 0x00000141 (decimal:  321)
  VA 66: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000081 (decimal:  129)
  VA 67: 0x00000050 (decimal:   80) --> VALID in SEG1: 0x0000012f (decimal:  303)
  VA 68: 0x00000029 (decimal:   41) --> VALID in SEG0: 0x000000a2 (decimal:  162)
  VA 69: 0x0000000f (decimal:   15) --> VALID in SEG0: 0x00000088 (decimal:  136)
  VA 70: 0x0000001d (decimal:   29) --> VALID in SEG0: 0x00000096 (decimal:  150)
  VA 71: 0x0000004c (decimal:   76) --> VALID in SEG1: 0x0000012b (decimal:  299)
  VA 72: 0x00000057 (decimal:   87) --> VALID in SEG1: 0x00000136 (decimal:  310)
  VA 73: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x0000007d (decimal:  125)
  VA 74: 0x0000003d (decimal:   61) --> VALID in SEG1: 0x0000011c (decimal:  284)
  VA 75: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x0000007d (decimal:  125)
  VA 76: 0x00000047 (decimal:   71) --> VALID in SEG1: 0x00000126 (decimal:  294)
  VA 77: 0x00000021 (decimal:   33) --> VALID in SEG0: 0x0000009a (decimal:  154)
  VA 78: 0x00000058 (decimal:   88) --> VALID in SEG1: 0x00000137 (decimal:  311)
  VA 79: 0x00000062 (decimal:   98) --> VALID in SEG1: 0x00000141 (decimal:  321)
  VA 80: 0x00000032 (decimal:   50) --> SEGMENTATION VIOLATION (SEG1)
  VA 81: 0x00000063 (decimal:   99) --> VALID in SEG1: 0x00000142 (decimal:  322)
  VA 82: 0x0000001e (decimal:   30) --> VALID in SEG0: 0x00000097 (decimal:  151)
  VA 83: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000080 (decimal:  128)
  VA 84: 0x0000003b (decimal:   59) --> VALID in SEG1: 0x0000011a (decimal:  282)
  VA 85: 0x00000003 (decimal:    3) --> VALID in SEG0: 0x0000007c (decimal:  124)
  VA 86: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x0000008c (decimal:  140)
  VA 87: 0x00000028 (decimal:   40) --> VALID in SEG0: 0x000000a1 (decimal:  161)
  VA 88: 0x0000003d (decimal:   61) --> VALID in SEG1: 0x0000011c (decimal:  284)
  VA 89: 0x0000000f (decimal:   15) --> VALID in SEG0: 0x00000088 (decimal:  136)
  VA 90: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x0000007d (decimal:  125)
  VA 91: 0x00000056 (decimal:   86) --> VALID in SEG1: 0x00000135 (decimal:  309)
  VA 92: 0x0000001f (decimal:   31) --> VALID in SEG0: 0x00000098 (decimal:  152)
  VA 93: 0x0000005f (decimal:   95) --> VALID in SEG1: 0x0000013e (decimal:  318)
  VA 94: 0x00000059 (decimal:   89) --> VALID in SEG1: 0x00000138 (decimal:  312)
  VA 95: 0x00000025 (decimal:   37) --> VALID in SEG0: 0x0000009e (decimal:  158)
  VA 96: 0x0000002e (decimal:   46) --> SEGMENTATION VIOLATION (SEG0)
  VA 97: 0x00000034 (decimal:   52) --> SEGMENTATION VIOLATION (SEG1)
  VA 98: 0x00000040 (decimal:   64) --> VALID in SEG1: 0x0000011f (decimal:  287)
  VA 99: 0x0000003b (decimal:   59) --> VALID in SEG1: 0x0000011a (decimal:  282)
```

5. Can you run the simulator such that no virtual addresses are valid?

Yes, by simply set both limits to 0…

python3 segmentation.py -a 128 -p 512 -l 0 -L 0 -n 10 -c

```
ARG seed 0
ARG address space size 128
ARG phys mem size 512
Segment register information:
  Segment 0 base  (grows positive) : 0x000001b0 (decimal 432)
  Segment 0 limit                  : 0
  Segment 1 base  (grows negative) : 0x00000184 (decimal 388)
  Segment 1 limit                  : 0
Virtual Address Trace
  VA  0: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000033 (decimal:   51) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000064 (decimal:  100) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000026 (decimal:   38) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x0000003d (decimal:   61) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x0000004a (decimal:   74) --> SEGMENTATION VIOLATION (SEG1)
  VA  8: 0x00000074 (decimal:  116) --> SEGMENTATION VIOLATION (SEG1)
```

VA 9: 0x00000040 (decimal: 64) --> SEGMENTATION VIOLATION (SEG1)

Chapter 17

1. First run with the flags -n 10 -H 0 -p BEST -s 0 to generate a few random allocations and frees. Can you predict what alloc()/free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?

alloc(): With BEST fit, the allocator searches the entire free list for the smallest chunk that fits the request.
All free() operations return 0 (success), and without coalescing, freed chunks are simply added back to the list sorted by address.

I observed the free list grows longer and more fragmented, making future allocations slower and potentially impossible (despite having enough total free space).

```
vm-freespace % python3 malloc.py -n 10 -H 0 -p BEST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[3] = Alloc(8) returned 1008 (searched 4 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[4] = Alloc(2) returned 1000 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[5] = Alloc(7) returned 1008 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

2. How are the results different when using a WORST fit policy to search the free list (-p WORST)? What changes?

BEST fit:

Creates many tiny unusable slivers (1-byte chunks)
Preserves larger chunks for future use
Final free list: [sz:1, sz:5, sz:1, sz:84]
Still has one large chunk (84) available

WORST fit:

Creates many medium-sized chunks
Continuously whittles down the largest chunk
Final free list: [sz:3, sz:5, sz:8, sz:8, sz:67]
No tiny slivers, but the large chunk shrinks from 84→76→74→67

```
python3 malloc.py -n 10 -H 0 -p WORST  -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy WORST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[3] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]
Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ]
ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1026 sz:74 ]
ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
```

Free List [Size 5]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:8][addr:1033 sz:67]

3. What about when using FIRST fit (-p FIRST)? What speeds up when you use first fit?

FIRST fit can terminate search early when an available space (big enough for alloc)is being found

BEST/WORST: O(n) - must examine every chunk to find best/worst
FIRST: O(n) worst case, but O(1) best case when first chunk fits.

In terms of fragmentation pattern, FIRST produces the same result as BEST here. Also, there is no need to track "best so far" or scan the entire list, so memory wise is more efficient.

malloc.py -n 10 -H 0 -p FIRST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [Size 1]: [addr:1003 sz:97]
Free(ptr[0])
returned 0
Free List [Size 2]: [addr:1000 sz:3][addr:1003 sz:97]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [Size 2]: [addr:1000 sz:3][addr:1008 sz:92]
Free(ptr[1])
returned 0
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:92]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]
Free(ptr[2])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]
ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]
Free(ptr[3])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]
ptr[4] = Alloc(2) returned 1000 (searched 1 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]
ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1015 sz:1][addr:1016 sz:84]

4. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (-l ADDRSORT, -l SIZESORT+, -l SIZESORT-) to see how the policies and the list orderings interact.

The list ordering dramatically affects search performance: BEST fit is fastest with SIZESORT+ (O(k) early termination since first fit = best fit) but slowest with SIZESORT- (must scan entire list)

WORST fit is fastest with SIZESORT- (O(1) since first chunk = largest) but slowest with SIZESORT+; and FIRST fit becomes a chameleon that mimics BEST with SIZESORT+ and WORST with SIZESORT-.

With ADDRSORT, all three policies must search through the list (BEST/WORST always O(n), FIRST variable O(k)), but fragmentation patterns differ—ADDRSORT concentrates fragments at low addresses, SIZESORT+ keeps small fragments at list front, and SIZESORT- keeps large chunks immediately accessible.

So pairing BEST+SIZESORT+ or WORST+SIZESORT- creates optimal O(k) or O(1) performance, but maintaining sorted order during free() operations adds overhead, so the "best" combination depends on whether to optimize for allocation speed, deallocation speed, or fragmentation control.

5. Coalescing of a free list can be quite important. Increase the number of random allocations (say to -n 1000). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the -C flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?
1000 allocation without -C:

Free List [Size 31]: [addr:1000 sz:2][addr:1002 sz:1][addr:1006 sz:1][addr:1007 sz:1][addr:1008 sz:5][addr:1013 sz:1][addr:1014 sz:1][addr:1015 sz:1][addr:1016 sz:5][addr:1021 sz:1][addr:1022 sz:3][addr:1031 sz:1][addr:1032 sz:2][addr:1034 sz:3][addr:1037 sz:4][addr:1041 sz:1][addr:1042 sz:2][addr:1052 sz:1][addr:1053 sz:6][addr:1059 sz:2][addr:1061 sz:1][addr:1068 sz:1][addr:1069 sz:3][addr:1072 sz:5][addr:1077 sz:3][addr:1080 sz:1][addr:1081 sz:5][addr:1086 sz:3][addr:1089 sz:5][addr:1094 sz:2][addr:1096 sz:4]

With -C:
Free List [Size 1]: [addr:1002 sz:98]

Coalescing REQUIRES ADDRSORT to work effectively. With ADDRSORT: Adjacent free chunks appear next to each other in the list (addr:1002, addr:1003), making them easy to detect and merge.

6. What happens when you change the percent allocated fraction -P to higher than 50? What happens to allocations as it nears 100? What about as the percent nears 0?

```
python3 malloc.py -n 10 -H 0 -p BEST -s 0 -P 51 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 51
allocList
compute True
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]
ptr[2] = Alloc(3) returned 1000 (searched 3 elements)
Free List [ Size 2 ]: [ addr:1003 sz:5 ][ addr:1008 sz:92 ]
Free(ptr[2])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]
ptr[3] = Alloc(10) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1018 sz:82 ]
Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:10 ][ addr:1018 sz:82 ]
ptr[4] = Alloc(8) returned 1008 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:2 ][ addr:1018 sz:82 ]
Free(ptr[4])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:82 ]
```

```
python3 malloc.py -n 10 -H 0 -p BEST -s 0 -P 100 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 100
allocList
compute True
ptr[0] = Alloc(8) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1008 sz:92 ]
ptr[1] = Alloc(3) returned 1008 (searched 1 elements)
```

```
Free List [ Size 1 ]: [ addr:1011 sz:89 ]
ptr[2] = Alloc(5) returned 1011 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1016 sz:84 ]
ptr[3] = Alloc(4) returned 1016 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1020 sz:80 ]
ptr[4] = Alloc(6) returned 1020 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1026 sz:74 ]
ptr[5] = Alloc(6) returned 1026 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1032 sz:68 ]
ptr[6] = Alloc(8) returned 1032 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1040 sz:60 ]
ptr[7] = Alloc(3) returned 1040 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1043 sz:57 ]
ptr[8] = Alloc(10) returned 1043 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1053 sz:47 ]
ptr[9] = Alloc(10) returned 1053 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1063 sz:37 ]

python3 malloc.py -n 10 -H 0 -p BEST -s 0 -P 1 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 1
allocList
compute True
ptr[0] = Alloc(5) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1005 sz:95 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:5 ][ addr:1005 sz:95 ]
ptr[1] = Alloc(2) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1002 sz:3 ][ addr:1005 sz:95 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:95 ]
ptr[2] = Alloc(9) returned 1005 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1014 sz:86 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]
ptr[3] = Alloc(2) returned 1000 (searched 4 elements)
Free List [ Size 3 ]: [ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]
Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]
ptr[4] = Alloc(4) returned 1005 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1009 sz:5 ][ addr:1014 sz:86 ]
Free(ptr[4])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:86 ]
```

The -P flag controls the allocation-to-free ratio and dramatically affects memory behavior: at

P=100 (all allocations), no frees occur so the heap steadily depletes (100→92→89→84 bytes) with the free list staying at exactly 1 chunk (O(1) search), eventually leading to out-of-memory as the program never releases anything.

At P=1 (mostly frees), the opposite problem emerges: the free list explodes in size (growing from 2→5 fragments in just 10 operations) with severe fragmentation creating many tiny unusable chunks (sz:2, sz:3, sz:4, sz:5) while the large chunk (sz:86) sits idle, and search costs increase linearly with list length—this simulates aggressive alloc/free patterns without coalescing.

At P=51 (balanced), the free list grows moderately (to 5 fragments) and memory usage fluctuates naturally.

7. What kind of specific requests can you make to generate a highlyfragmented free space? Use the -A flag to create fragmented free lists, and see how different policies and options change the organization of the free list.

```
python3 malloc.py -A +10,+10,+10,+10,+10,-0,-2,-4 -p BEST -l ADDRSORT -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList +10,+10,+10,+10,+10,-0,-2,-4
compute True
ptr[0] = Alloc(10) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1010 sz:90 ]
ptr[1] = Alloc(10) returned 1010 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1020 sz:80 ]
ptr[2] = Alloc(10) returned 1020 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1030 sz:70 ]
ptr[3] = Alloc(10) returned 1030 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1040 sz:60 ]
ptr[4] = Alloc(10) returned 1040 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1050 sz:50 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:10 ][ addr:1050 sz:50 ]
Free(ptr[2])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:10 ][ addr:1020 sz:10 ][ addr:1050 sz:50 ]
Free(ptr[4])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:10 ][ addr:1020 sz:10 ][ addr:1040 sz:10 ][ addr:1050 sz:50 ]
```

Even with the -C flag, coalescing only merges adjacent free chunks. Here, free chunks at addr:1000, addr:1020, and addr:1040 are not adjacent because allocated blocks (ptr[1] at 1010-1019 and ptr[3] at 1030-1039) separate them.

```
vm-freespace % python3 malloc.py -A +10,+10,+10,+10,+10,-0,-2,-4 -p BEST -l ADDRSORT -C -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce True
numOps 10
range 10
percentAlloc 50
allocList +10,+10,+10,+10,+10,-0,-2,-4
compute True
ptr[0] = Alloc(10) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1010 sz:90 ]
ptr[1] = Alloc(10) returned 1010 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1020 sz:80 ]
ptr[2] = Alloc(10) returned 1020 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1030 sz:70 ]
ptr[3] = Alloc(10) returned 1030 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1040 sz:60 ]
ptr[4] = Alloc(10) returned 1040 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1050 sz:50 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:10 ][ addr:1050 sz:50 ]
Free(ptr[2])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:10 ][ addr:1020 sz:10 ][ addr:1050 sz:50 ]
Free(ptr[4])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:10 ][ addr:1020 sz:10 ][ addr:1040 sz:60 ]
```